

Developing a scalable, performant and maintainable Game AI System, as an extension for the Unity3D Engine

Media Informatics
Department of Electrical Engineering and Computer Science
Technische Hochschule Lübeck

Bachelor Thesis Submitted by Nikodem Grzonkowski
4th October 2021 - 4th January 2022

Abstract

Department of Electrical Engineering and Computer Science

Bachelor of Science (B.Sc.)

**Developing a scalable, performant and maintainable Game AI System, as an extension
for the Unity3D Engine**

by Nikodem Grzonkowski

Common artificial intelligence (AI) patterns mostly cover specific areas of AI systems (AIS), such as either decision-making, action management or data storage not equally well if each area is implemented in the first place. This oftentimes results in convoluted patterns, lacking the requirements for a maintainable, performant and scalable generic AIS, as it can be observed with AI asset packs available at the Unity3D Asset Store. The main subjects of this thesis are researching, evaluating and solving the discovered issues of established AI patterns in theory and asset packs in practise, by benchmarking said asset packs with tools provided by the Unity3D Engine and finally developing a generic AIS prototype as an extension to this game engine using its designated programming language C#.

Table of Contents

Abstract	i
Table of Contents	ii
1 Introduction.....	5
1.1 Prerequisite.....	5
1.2 Definition of Game AI	5
1.3 Motivation.....	5
1.4 Goal.....	5
1.5 Organization.....	6
1.6 Unity3D Keywords.....	6
2 Requirements	7
2.1 Scalability.....	7
2.2 Performance	8
2.3 Maintainability	8
3 Research and Evaluation of AI Patterns	9
3.1 Finite State Machines	9
3.2 Decision Trees.....	10
3.3 Behavior Trees	12
3.4 Rule-Based Systems.....	13
3.5 Goal Oriented Behavior.....	14
3.6 Fuzzy Logic.....	15
3.7 Blackboards.....	15
3.8 Action Management.....	16
3.9 Scheduling.....	17

3.10	Conclusion.....	18
4	Evaluation of AI Asset Packs.....	18
4.1	Evaluation Process	20
4.2	████████████████████.....	22
4.3	████████████████████.....	28
4.4	████████████████████.....	30
4.5	████████████████████.....	39
4.6	████████████████████.....	48
4.7	Conclusion.....	50
5	Prototype Development.....	51
5.1	Settings.....	52
5.1.1	State Settings.....	53
5.1.2	Condition Settings.....	54
5.1.3	Action Settings.....	56
5.1.4	Decider Settings.....	57
5.2	Mapping Types.....	60
5.3	Blackboards, Actions and Engines	63
5.4	Descriptors	70
5.5	Decision System.....	71
5.6	OneAIEntity	74
5.7	Graph Editor.....	76
5.8	Code Generation.....	79
6	Prototype Evaluation.....	80
7	Conclusion and Outlook.....	90

Acknowledgments.....	91
Appendix A - List of Figures	92
Appendix B - List of Listings.....	94
Appendix C - List of Tables.....	96
Appendix D - List of External Frameworks	96
Appendix E - List of Required Tools.....	96
Bibliography	97

1 Introduction

1.1 Prerequisite

An intermediate to advanced expertise in object-oriented programming, specifically in the programming language C# [Tool-1] is expected by the reader.

Terms such as reflection, inheritance, event-driven architecture (EDA), garbage collection (GC), generics, etc., are without further explanation mentioned in this thesis.

Knowledge about artificial intelligence, the Unity3D Engine [Tool-2] and video games in general is recommended but not mandatory.

1.2 Definition of Game AI

Artificial intelligence (AI) in games is bound to hardware and time limitations and therefore unlike academic AI focuses on the illusion of intelligence, rather than simulating AI as close to the real world as possible [1].

The term AI nowadays is oftentimes wrongly associated exclusively with neural networks or machine learning, although it is true that said fields are part of the entire AI spectrum.

1.3 Motivation

Building game AI for a game project from scratch is especially difficult and time consuming. For that reason, it is recommended to either integrate an existing generic AI system (AIS) or build one, which is at best compatible with every game genre and is reusable for many game projects to come.

Investigating the Unity Asset Store, results in many advertised AI asset packs either specializing on certain game genres or lacking criteria for a generic and performant AIS, leaving room for improvement, which is the core motivation for this thesis.

1.4 Goal

The goal of this thesis is to emphasize, that popular AI patterns have difficulties to meet the requirements for a scalable, maintainable and performant generic AIS, unless modified. This is highlighted by examining and evaluating AI frameworks, as well as AI asset packs available at the Unity3D Asset Store, finally solving the observed issues by building an AIS prototype for the Unity3D Engine.

1.5 Organization

Chapter 2 defines requirements for a generic AIS as a foundation for the research and evaluation for established AI patterns in Chapter 3 and existing AISs in Chapter 4.

Including the observations (established issues and improvement proposals) from Chapter 3 and 4, the architecture of the AIS prototype is described in Chapter 5. Following with Chapter 6 the built AIS prototype is evaluated to inspect if the requirements are met and the issues solved, finally summarizing the results in Chapter 7.

1.6 Unity3D Keywords

Keyword	Explanation
GameObject	Fundamental object in Unity, acting as a container for Components [2].
Prefab	Create, configure and store a GameObject as a reusable template asset to instantiate in a Scene [3].
ScriptableObject	Data container to save large amounts of data, independent of class instances and Scenes [4].
Component	Defines a behavior for a GameObject and acts as a base class for the MonoBehaviour class [5].
MonoBehaviour	Provides hooks for useful events, such as Start and Update [6].
Scene	Assets, representing game levels and environments with their respective GameObjects [7].
Inspector	View and edit properties for almost everything related to the Unity3D engine, for example GameObjects, Components and in-Editor settings [8].
Asset Store	A marketplace for Unity3D developers, offering asset packs created by other Unity3D developers (hobbyists, professionals, companies, etc.) [9].

Table 1-1: Unity3D Keywords

2 Requirements

Before any evaluation or development process can begin, first the requirements of a generic game AIS in the context of this thesis must be defined, which is the subject of this chapter.

2.1 Scalability

The generic AIS should support a wide range of Unity versions, preferably starting with the minimum supported version in the Asset Store (2018.4.0f1), up to the newest available Unity long time support (LTS) versions. Furthermore, the generic AIS must at least be compatible with the following operating systems (OS):

- Windows
- iOS
- macOS
- Android
- Linux

The generic AIS must be free of restrictions for creating custom AI agents and behaviors, meaning that no pre-defined template is mandatory, such as a (3D) mesh, rig and animation controller.

Custom AI behavior can be added at any time while working in the Unity project by offering expansion of the core AIS modules via inheritance. This includes implementing custom decision-making, action execution and data containers, able to interact with the (modular) core AIS, if required.

Providing said conditions should theoretically cover the basis of a generic AIS, able to support AI behaviors for any game genre and agile adaptations during the game AI development process.

2.2 Performance

Minimizing performance costs as much as technically possible is a crucial topic for game development in general. Many game relevant algorithms require execution time for and access to the central processing unit (CPU), random access memory (RAM) or graphics processing unit (GPU), including but not limited to:

- Rendering (Textures, Models, Shaders, etc.)
- Animation
- Physics Calculation
- Audio System
- AI

For that reason, the core modules of the generic AIS must provide stable performance levels for lots of AI entities in a dense area, using as little CPU, GPU and RAM as technically feasible. Though it is the responsibility of the AI developer to manage the performance costs for his or her own custom built AI algorithms, the generic AIS must implement efficient and fast algorithms as a foundation for any custom-built AI behavior.

This concludes, that not every module has to be executed every frame during the lifetime of a game, but rather when relevant which can be done via an EDA.

2.3 Maintainability

The more complicated the (growing) AI behavior gets, the more important is the overview and maintainability for it. This leads to a system, which must offer a user experience of high quality by providing a graphical user interface (GUI) for the AI maintenance, which is at best self-explanatory in terms of navigation, interaction and naming of functionalities.

Potential actions and decisions an AI entity can make are to be visually displayed in a GUI, which preferably offers a debugging system, showcasing the current decisions made by an AI instance at runtime. Furthermore, adding, removing and modifying the AI behavior within the Unity project must be fast and easy to access for the AI designer and developer.

The same rules apply to creating custom from the generic AIS inherited AI code by providing options to automatically generate scripts form pre-defined template files, saving time writing boilerplate code.

Related to the scalability requirement, no limitations or exclusions in the form of models, rigs, animations, scripts, etc. should be a consequence of integrating and using this generic AIS.

3 Research and Evaluation of AI Patterns

This chapter covers the research and evaluation of established AI patterns based on the requirements highlighted in Chapter 2, focusing rather on the application of the patterns for a generic AIS, than the in-depth explanation.

3.1 Finite State Machines

“Finite-state machines are a model of computation with limited amount of memory known as a state. Each machine has only a finite number of possible states (for instance, wander or patrol). A transition function determines how the state changes over time, according to the inputs to the finite-state machine” [10, p. 509]

Furthermore, a state or state transition can group and return a collection of actions for the entry, loop and exit of the state to alter the environment of the game. [11]

Figure 3-1 pictures an example for a finite state machine (FSM), describing an AI behavior for an agent that searches, follows and attacks an enemy visualized with states and their respective transitions.

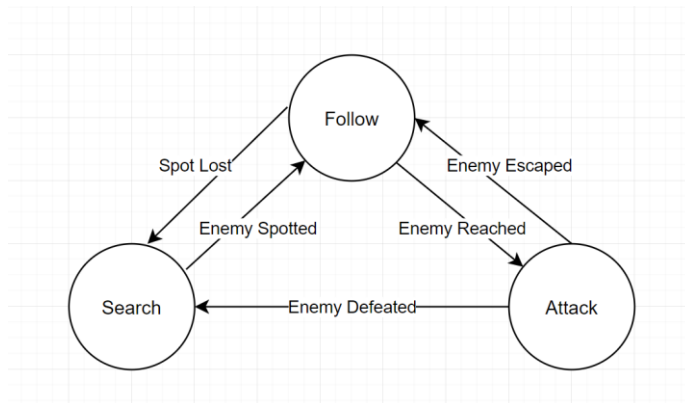


Figure 3-1: Simple FSM Example

Adding a state requires to setup a transition to the next desired state, hence it is not possible to reach any other state from the current state, without a pre-defined transition. Considering this special case, the FSM growth can be described as $m \cdot n$ total transitions, where m = transitions per state and n = the state amount. For the worst-case scenario, where every in the FSM existing state must be reachable from every other state ($m = n$), the growth can be described in the exponential Big O Notation $O(n^2)$ [12]. While this is not likely to happen if an experienced and efficient AI designer works with a FSM, this maintainability issue persists for sophisticated AI behaviors, which require lots of states and transitions. For that reason, the **requirement maintainability** defined in Section 2.3 is **not met** for a flat FSM.

Using a hierarchical FSM is an improvement but does not solve this issue on a layer basis, since each layer represents a flat FSM.

The performance of FSMs is $O(1)$ in memory and $O(m)$ per time [11, p. 320], where m = transitions per state, resulting in a linear growth in performance costs, which is **acceptable** in regard to the **performance requirement** mentioned in Section 2.2.

Since the actions are accessible directly on the states, the action execution is quite efficient, only in the case of the transition checks being called separately. Other than that, mapping and injecting runtime and action specific data is not defined in the FSM architecture but is mandatory for the scalability requirement as stated in Section 2.1.

Since the **FSM pattern lacks the maintainability and scalability requirements**, it is not recommended to be implemented in a generic AIS. The upside of using FSMs though, is the ease of debugging an AI agent, since the current active state is clearly visualized and potentially easy to trace. Considering this, a state which shows the AI agent's current active behavior and potentially having an impact in the next decision cycle, is a useful feature for a generic AIS.

3.2 Decision Trees

“A decision tree is made up of connected decision points, also called choices or nodes. The tree has a starting decision, its root. For each decision, starting from the root, one of a set of options is chosen. These choices lead either to further decision, or to a final action”

[11, p. 300]

The branching of decision trees is built either binary, where a node can only subdivide in at max two other nodes (see Figure 3-2) or not limited at all (see Figure 3-3) [11].

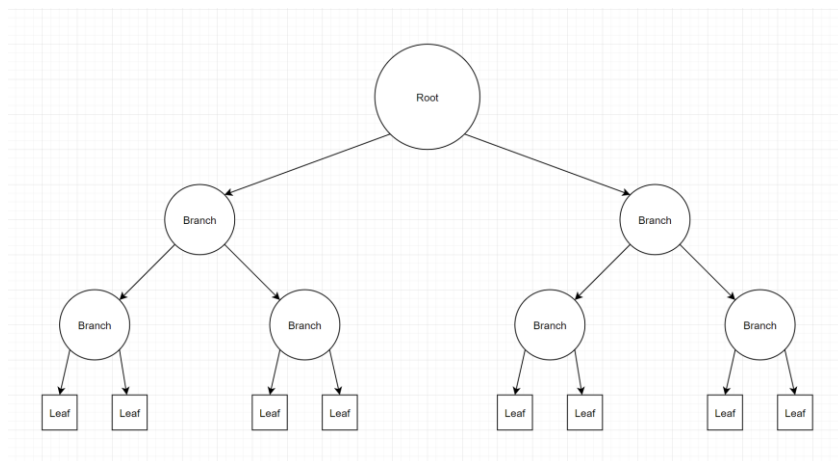


Figure 3-2: Simple Binary Decision Tree Example

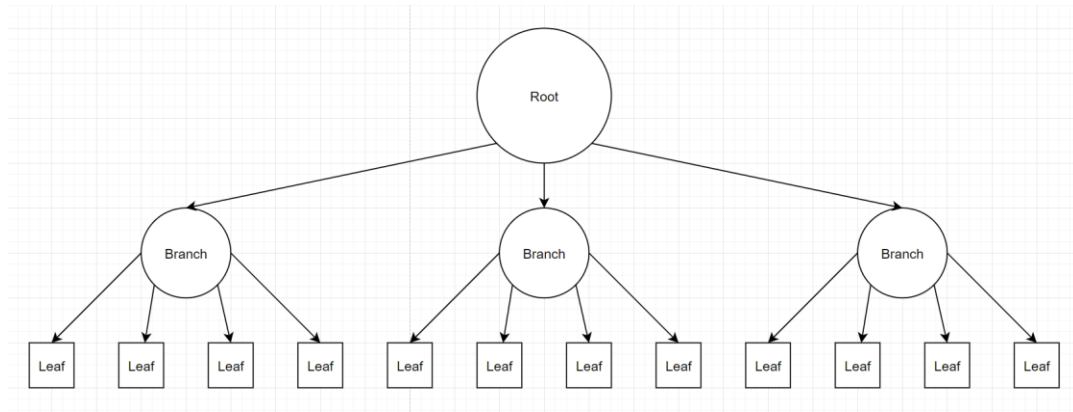


Figure 3-3: Simple Non-Limited Decision Tree Example

For maintainability and memory reasons it is not recommended to implement a binary decision tree, since more branch nodes must be created to reach a leaf node, compared to a non-limited decision tree. Although the advantages of a binary tree being the hierarchical consistency and a more balanced decision tree, resulting in less conditional checks and better performance. On the other hand, a binary decision tree can also be built using a non-limited decision tree implementation, leaving both options to be chosen by the AI designer.

The definition of a balanced tree describes that the tree has about the same number of leaves for each branch. The performance of a balanced tree is at best $O(\log(n))$, where n = the number of decision nodes in the tree. At worst a (balanced) decision tree traverses through each node and has a performance cost of $O(n)$ [11, p. 308].

Considering this, the **performance requirement** defined in Section 2.2 **is met** for the decision-making architecture, since $O(\log(n))$ is the second best growth in terms of saving performance, besides the constant $O(1)$ [12]. As for the action execution, the decision tree pattern must always traverse through its branches in order to execute them, which is an inefficient implementation, as it does not provide the single responsibility principle [13]. This can be avoided by separating decision-making from action execution, by notifying action manager classes with the resulting leaf nodes and thus prevent redundant traversals, providing a modular and scalable architecture. Therefore, the **scalability requirement** from Section 2.1 **is not met** for this pattern.

Unlike the FSM pattern highlighted in Section 3.1, a decision tree grows linear $O(n)$ in terms of its branch and leaf nodes, since it is not forced to act from its last resulted leaf node and additionally is always run from its root node. For that reason, the decision tree pattern is easy to overview and maintain and therefore **meets the maintainability requirement** defined in Section 2.2.

In conclusion, the decision tree pattern provides a valid decision-making algorithm for a generic AIS, as long as it is separated from the action management and data injection.

3.3 Behavior Trees

The behavior tree pattern, as illustrated in Figure 3-4 consists of four general task types [11]:

- Composites
- Conditions
- Decorators
- Actions

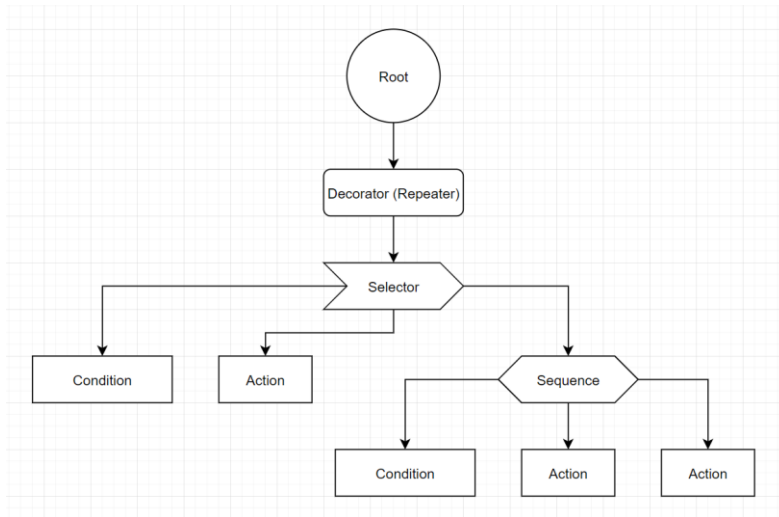


Figure 3-4: Simple Behavior Tree Example

Composite tasks are representative for branches in the behavior tree and keep track of child tasks, which in return can also be a task of type composite, condition, decorator, or action [11].

Common composite tasks are selectors and sequences. Both composite types run their children one after another and decide if they should continue or stop, depending on the result of the last run child task. A selector will immediately stop and return a successful result, once any child node finishes with success, unlike a sequence composite which invalidates and stops, once a failed child task is found. There are a variety of other composite tasks, that handle the execution of their underlying children and it is possible to implement a custom composite task via inheritance.

Both condition and action tasks are found in the leaf nodes of the behavior tree. A condition task tests some properties of the data it is given and returns a result, in which the parent composite task can decide how to proceed. An action task alters the state of the game, by for example executing an animation or moving the designated agent to its desired location.

The decorator task behaves like a wrapper class for a single child task, with the purpose of modifying it in some shape or form. As demonstrated in Figure 3-4, a simple example for a decorator class is a repeater, which repeats the underlying child until a certain interrupt or fail condition is met.

Behavior trees share a similar structure to decision trees and therefore have the same performance costs for best and worst-case scenarios, which are $O(\log(n))$ at best and $O(n)$ at worst (see Section 3.2) [11, p. 346]. Another commonality with the decision tree is, that the behavior tree pattern always must traverse through its branch nodes, in order to execute an action task, thus wasting performance. As stated in Section 3.2, it is a more performant and modular approach to separate action management and data injection from the decision-making process.

The behavior tree pattern offers a modular and expandable approach for custom task creation via inheritance but similar to the decision tree pattern is convoluted regarding action execution and decision-making. As a result, this pattern does **meet** the **performance** and **scalability requirements** (see Section 2.1 and 2.2) for the **decision-making** system, but **not** for **action execution** and **data management**.

Furthermore, it is an ineffective approach to use condition tasks as leaf nodes to handle the traversal of a tree, forcing the AI designer to use a sequence composite task with a condition task and another composite task as a child to prevent the traversal of a sub tree. As a result, redundant nodes are added to the tree, which could be avoided by providing an option to attach a condition task to any task type. Solving this inconvenience, this AI pattern **meets** the **maintainability requirement** defined in Section 2.3.

3.4 Rule-Based Systems

The rule-based system (RBS) contains a database, that offers knowledge of the game world to the AI and a set of if-then rules, that lead to the desired decision [11].

RBSs offer multiple ways to interpret valid and invalid condition results, by using rule arbitration, adding flexibility to the decision-making process [11]. For instance, it may be the case to pick only the first valid result of a rule set (first applicable) or collect and execute all valid results. In addition, it is possible to cache the last recently used ruleset to save time in the next decision cycle.

As for the performance costs, the RBS has a growth of $O(nm)$, where n = items in the database and m = clauses to check [11, p. 447], leading to $O(n^2)$ at worst for $n = m$ or $O(n)$ at best. The worst case can be avoided, if the required data is requested directly and not searched, resulting in a similar performance cost growth to the FSM pattern (see Section 3.1) and therefore **meeting** the **performance requirement** defined in Section 2.2.

The **scalability requirement** (see Section 2.1) of the RBS **can be achieved**, by adding rules to rulesets as custom encapsulated objects, which contain an overwritable method returning either true or false. As for the database and data injection, it is recommended to use a modular approach and define specific databases for rule objects, rather than a single database that stores all information, as it is further established in Section 3.7. Additionally, actions must be built as separate encapsulated objects as well, to be dynamically assigned to rule or ruleset objects, preventing a hardcoded architecture [13].

The **maintainability requirement** outlined in Section 2.3 is **not met** for this pattern, due to the fact that the rules are stored in a flat list and no visual representation of an RBS is provided by default.

“They remain a fairly uncommon approach, partly because similar behaviors can almost always be achieved in a simpler way using decision trees or state machines” [11, p. 431]

In conclusion, this AI pattern lacks the maintainability requirement and is worse in terms of performance compared to the patterns in Section 3.2 and 3.3, but offers an interesting approach to interpret decision results using the rule arbitration feature, which potentially can be combined with other AI patterns.

3.5 Goal Oriented Behavior

“Goal-oriented behavior is a blanket term that covers any technique taking into account goals or desires.” [11, p. 406]

The goal-oriented behavior (GOB) is not in particular a decision-making algorithm, but rather an approach in how an agent uses the decision system to achieve its pre-defined goals.

For example, an agent needs to stay saturated and hydrated, while trying to reach a faraway destination point. Those goals/desires are represented as *hunger*, *thirst* and *keep moving*, each stored as a number (integer, or float). For every desire, certain actions are defined that either drain or satisfy said desires, for instance:

- Move to destination action = -2 keep moving, +1 hunger, +2 thirst
- Drink from bottle action = +1 keep moving, -4 thirst
- Eat food action = +2 keep moving, +1 thirst, -4 hunger

Depending on the desire's priority and its satisfaction level, the next decision may stop a running action and start another one, which fulfills the most needed desire, maintaining the balance for all desires.

In conclusion the GOB **neither validates nor invalidates** the **requirements** defined in Chapter 2, since it is built on top of an existing AI pattern.

3.6 Fuzzy Logic

“Fuzzy logic [...], enables a computer to reason about linguistic terms and rules in a way similar to humans. Concepts like “far” or “slightly” are not represented by discrete intervals, but by fuzzy sets, enabling values to be assigned to sets to a matter of a degree – a process called fuzzification” [1, p. 416]

Since the basic fuzzy logic pattern does not represent any (hierarchical) decision management, it is not recommended as a top-level AI pattern, but rather as a useful extension to an existing AI pattern to simulate intuitive decision-making. The fuzzy logic pattern is basically a more sophisticated version of the RBS described in Section 3.4 but due to its specialization does **not meet the requirements** defined in Chapter 2.

3.7 Blackboards

“A blackboard system isn't a decision making system in its own right. It is a mechanism for coordinating the actions of several decision makers.” [11, p. 461]

Actions that need to be performed by an agent oftentimes need information of the current environment, such as other agents, world objects or the agent it is running on. For example, an agent that needs to flee to designated destination points on the world map with obstacles and chasers to avoid, which only exist and change at runtime.

Blackboards describe a wrapper class, which contains all necessary information required by the executing action. During the actions lifetime the blackboards are managed by a designated class and fed into the action as a parameter. The action then has the option to read or modify the contents of the acquired blackboard.

Using a single (global) blackboard for all possible action types results in poor scalability, since no inheritance is provided and in addition not every referenced data type is required by every action type, wasting memory and most likely performance as well [13]. This leads to an implementation form that requires a specific (local) blackboard type, mapped to its corresponding action type, contained and handled within a specific manager class.

This is complicated to build but solves the scalability issue and offers the option to add custom action and blackboard types to the AIS. On the other hand, the negative aspect of this approach, is that the AI developer is forced to create 3 separate classes (action, blackboard, manager) for a new action type, as a result decreasing the maintainability for AI programmers. To solve this issue, a script generation tool must be offered by the AIS, creating the dependent classes from template files, which contain the relevant boilerplate code.

Since the blackboard architecture primarily stores data, **no significant performance costs** are produced. Building a local blackboard architecture for (action) data-injection in this modified form **meets the scalability requirement** (see Section 2.1), but in return adds **difficulties** for the **maintainability requirement** (see Section 2.3). Although, this cannot be avoided without losing scalability, since the amount of action types for any game is unknown.

3.8 Action Management

The action management pattern examined in this section describes a system, which handles the execution and lifetime of actions and is unlike the previous discussed patterns not particularly related to AI or decision-making and therefore represents an entire separate module with the specific task of action management only.

Some actions are more sophisticated, than a single function call and oftentimes need to repeat for an indefinite time, requiring a manager class, which offers mechanics such as:

- Detecting an actions state in its current lifecycle and reacting to it
 - Running, paused, initializing, finished
- Interrupting a running action
- Adding and removing actions
- Order of action execution

Looping through each action in the action manager costs $O(n)$ in performance and memory, where n = the amount of actions to execute, assuming every action costs $O(1)$ in performance and memory. Since this pattern does not cover any decision-making algorithms, it must be combined with another pattern, such as the ones mentioned in Section 3.1 through 3.4.

This results in not all possible actions of an AI behavior being stored in an action managers list, but rather added or removed dynamically depending on the last decision result. Considering this and that the most efficient performance cost growth of the examined decision-making pattern is $O(\log(n))$ at best and $O(n)$ at worst, this pattern **meets the performance requirement** as of Section 2.2.

The data-injection is not handled in the action manager pattern but is required for some actions that need to change the world state, based on the world input. Combining this pattern with the blackboard pattern mentioned in Section 3.7 would solve this issue by injecting a blackboard into the current executing action.

To find the relevant blackboard for the respective action, a specialized action manager must be provided, which is restricted to handle only specific (base) actions of a certain type, that in return only accept a compatible blackboard type.

As a standalone pattern the **maintainability requirement** (see Section 2.3) can **neither be validated or invalidated**, though if combined with the blackboard pattern, a code generation tool must be provided for the AI developers to prevent writing the same boilerplate code for each new action type.

The **scalability requirement** defined in Section 2.1 **can be achieved** by providing overwritable methods and encapsulated classes for actions, action managers and blackboards.

3.9 Scheduling

“A scheduling system manages which tasks get to run when. It copes with different execution frequencies and different task durations.” [11, p. 804]

The scheduling pattern works with a total time budget, which needs to be distributed across the next running actions in addition to the (different) frequencies the actions can be called [11]. Basically, a scheduler is a more sophisticated version of an action manager and similar to it is not related to AI and decision-making in particular.

Since the scheduling pattern offers more functionalities than the action manager pattern, more overhead in calculation is generated, but nevertheless outputs the same performance and memory cost growth $O(n)$, where n = action amount to execute. What is done better in comparison to the action manager is that the frequency between each action call can be set in a way that the actions are not called every frame and therefore performance is saved. Giving certain actions more time to perform per frame than others is crucial for a generic AIS as well, since not all actions share the same performance costs, for instance animation, pathfinding or spotting actions. Therefore, the **performance requirement** defined in Section 2.1 **is met** for this pattern even **more efficiently, than the action manager** examined in the previous Section 3.8.

Provided that the scheduling pattern is combined with a data-injection and decision-making pattern, it shares the **same requirement achievements** for **scalability** and **maintainability** (see Section 2.1 and 2.3) as the **action manager**.

3.10 Conclusion

In conclusion, there is no (AI) pattern, that meets all requirements defined in Chapter 2 for a generic AIS equally well. It is rather a **combination of multiple (modified) patterns** to form a generic AIS, such as the *action manager or scheduler* for action handling (see Section 3.8 and 3.9), the *decision tree or behavior tree* for decision-making (see Section 3.2 and 3.3) and finally the *blackboard pattern* (see Section 3.7) for data-injection.

Usually, agents in games maintain a decision for a certain amount of time, like chasing another agent until a certain state change occurs. Considering this, the decision-making pattern should offer multiple implementation forms for calling the decision-making algorithm, since it is not mentioned in the investigated patterns starting from Section 3.1 through 3.4 in how often the decision-making algorithm must be executed per frame or any other time interval. To solve this unclarity, a special scheduler (or action manager) must be built, that acts as a wrapper class for decision-making calls only, offering an EDA or polling technique to execute decisions.

Both polling and the EDA have pros and cons depending on the use case. For instance, it is not necessary to check if a certain object exists every frame, but to wait until the object emits an event once it appears in the game world. On the other hand, there might be a use case where an agent needs to follow a moving target in a fluent manner and cannot wait until the target emits a position changed event, which in addition would be less efficient performance wise in any case [11].

4 Evaluation of AI Asset Packs

This chapter covers the evaluation of AI asset packs available at the Unity3D Asset Store, based on the requirements defined in Chapter 2 in addition to the observations made in Chapter 3.

First, a curated list of the AISs available at the Unity3D Asset Store is created, since not every AIS is relevant for this thesis, for instance genre specific AISs such as racing or flight game AI. The targeted AISs are the ones, which claim to offer a generic architecture for every game genre, as well as customizable and expandable AI logic.

AI asset packs listed on the Asset Store are found with search terms such as “AI” [9] or “Behavior” [14] and can be ordered either by *relevance*, *popularity*, *rating*, *most favored*, *published date*, *name*, *price* or *recently updated*. While it is unknown how the order by algorithm works in detail, the most important ones for this evaluation are *relevance*, *popularity* and *rating*.

Following five AI systems are chosen and further examined in this thesis:

- [REDACTED] [15]
- [REDACTED] [16]
- [REDACTED] [17]
- [REDACTED] [18]
- [REDACTED] [19]

Due to the limited amount of time for this thesis, not all generic AISs available at the Unity Asset Store are evaluated. Systems worth mentioning, which could not be included in this thesis are *Ultimate AI System* [20], *Behavior (Game Creator 1)* [21], *Dynamic AI* [22], *AI Behavior* [23] and *Breadcrumb Ai* [24].

Moreover, the chosen AISs for evaluation are explained briefly in their functionality and code, due to it being out of scope for this thesis and in addition to all relevant information about the AI asset packs being located on their respective store page and documentation.

Each evaluated AI asset pack may not exist in the Asset Store, or may change its contents (description, documentation and the AIS itself) at a future time. In that case the evaluated asset packs can be found in the provided separate Unity benchmark test project, located within the USB device. Furthermore, the minimum supported Unity version in the Asset Store (2018.4.0f1), as well as other unknown subjects regarding the Unity3D Engine will likely change in the future.

4.1 Evaluation Process

The evaluation process steps applied to the chosen AI asset packs are defined as follows:

1. Integrating the AI asset pack into a for the AI evaluation designated Unity3D project.
2. Examining the documentation and sample Scenes to understand the overall structure, OS and Unity version compatibilities, usage of features, as well as a brief overview of its architecture.
3. Comparing the decision-making, action handling and data management algorithms of the AIS to the in Chapter 3 investigated AI patterns and evaluating their performance, maintainability and scalability based on the requirements described in Chapter 2.
4. Testing the AI asset pack with custom code and use cases, to either confirm or deny the established requirements with the following steps:
 - a. Create a separate (encapsulated) Unity Scene for the testing environment.
 - b. Create custom actions, which manipulate the runtime agent and the game world in some manner.
 - c. Create custom conditions to randomly succeed or fail.
 - d. Create custom data-containers for the custom actions and conditions to read from and write to.
 - e. Build a testing AI behavior with the custom created actions, conditions and data containers.
 - f. Benchmark multiple agents in one Scene at once for 30 seconds, each with 16 potential actions to reach, while using Unity's Profiler tool [25] as a measurement. Starting with 1, afterwards 10, then 100 and finally 1,000 agents at once, always counting the total decision and action execution count during the running benchmark tests.

To evaluate all AISs equally, the custom actions, data-containers and conditions reference the same implementation across all evaluated AISs. Moreover, all for the test irrelevant applications and Unity Editor windows, which could temper with the benchmark results are disposed of. Additionally, the Scene setup for each test is equal for all AISs in its structure and focuses on the AI performance costs only, meaning all redundant calculations, such as animations and renderings are excluded from the measurements as far as possible.

The hosting hardware, which benchmarks the AISs has following properties:

- OS: Microsoft Windows 10 Pro (10.0.19043)
- CPU: Intel® Core™ i7-7700HQ, Driver version (10.0.19041.546)
- RAM: 16GB, DDR4
- GPU: NVIDIA GeForce GTX 1070, Driver version (471.96)

The chosen Unity3D version for the testing project relevant for step 1 of the evaluation process is of version 2020.3.19f1 LTS.

The benchmarking action and condition classes perform a square root calculation over the same value, iterating over it 100 times to simulate an expensive operation [26]. In C# and Unity this can be achieved with the code snippet shown in Listing 4-1.

```

9      public static void WastePerformance()
10     {
11         double x = 1;
12         for (int i = 0; i < 100; i++)
13         {
14             x = Mathf.Sqrt(6);
15         }
16     }

```

Listing 4-1: Performance Heavy Method Simulation

To achieve a random condition pass of 50% in C# and Unity, a floating-point value of 0.5 is checked against a random generated floating-point number between 0 and 1.

To test the instance-based custom data injection per agent, a wrapper class is created (see Listing 4-2), containing primitive values which are tempered with during the Scene at runtime by the custom condition and action classes. This is achieved by defining integer values in the data container which get incremented by the accessing (condition or action) classes and is observed in Unity's Inspector when agents are selected individually, supported by the string values to visualize and confirm the uniqueness of the current selected agent.

```

6      public class CustomDataContainer
7      {
8          public int actionsCounter = 0, decisionsCounter = 0; < Serialized
9          public string actionsName = "", decisionsName = ""; < Serializab
10
11         public CustomSharedDataComponent customSharedDataComponent;
12     }

```

Listing 4-2: Custom Data Container

The *CustomSharedDataComponent* field is a reference to a Component, shared across all agent instances and stores the total global action and decision execution amount to oppose the performance costs to the total executions made.

If the AIS architecture enables it, the decision-making process is executed separately from the action execution in order to illustrate the contrast of time and memory consumed between the action and decision-making executions. Furthermore, the AI behavior setup for the 16 possible actions as per step 4.e and 4.f must be built as balanced as possible to ensure the same testing environment for every AI asset pack and to simulate a well-designed AI behavior, as mentioned in Section 3.2.

As per the Unity manual [25], the evaluation relevant profiler modules are briefly explained in Table 4-1.

Profiler Module	Category	Description
CPU	Scripts	How much time milli seconds (ms) your application spends on running scripts.
Memory	Object Count (OC)	The number of native object instances in your application.
Memory	GC Used Memory (GCUM)	The amount of memory used by the GC heap.
Memory	GC Allocated in Frame (GCAF)	The amount of memory allocated per frame on the GC heap.

Table 4-1: Profile Moduler Explanation

The recording window for each benchmark test is restricted to 600 frames, due to Unity's Profiler recording limit, in addition to its caused overhead for the memory and CPU hardware components. It is also worth mentioning, that the Unity Editor itself generates overhead for each module and category, especially visible for test cases with a small number of agents and method calls. All evaluation results, including the Profiler recordings are located in the benchmarking Unity project, saved on the provided USB device.

4.2 [REDACTED]

Evaluation process: Step 1

The integration of the asset pack [REDACTED] (BD) version 1.7 into the Unity testing project as mentioned in step 1 of the evaluation process works with no issues.

Evaluation process: Step 2 & 3

Based on the store page description [15] and documentation [27] BD is an AIS mainly built upon the behavior tree decision-making system as examined in Section 3.3.

Furthermore, it seems like there are no limitations regarding the supported OSs, only that special handling is required for UWP apps. The minimum required Unity version is 2017.4.1 and with that is below the minimum possible supported Unity version on the Asset Store (2018.4.0f1).

Unfortunately, as of the time this thesis is written, the AIS development studio does not provide sample (tutorial) files from their website, nor in the main AI asset pack for some reason. Although this being a minor inconvenience, the documentation and video tutorial series are explaining the usage of this AI asset pack clearly.

As illustrated in Figure 4-1 BD uses a node-based graph user interface (UI) in the Unity Editor to build AI behaviors in.

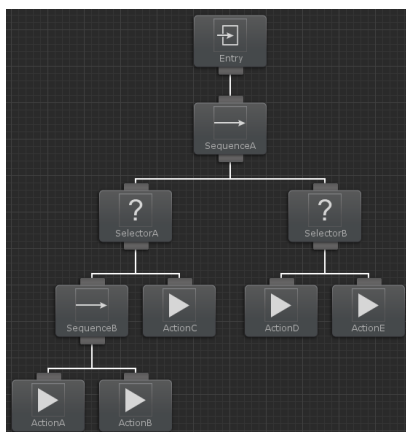


Figure 4-1: BD Graph UI Excerpt

The behavior tree is placed on a `GameObject` either as an instance in a Scene or as a Prefab, while the nodes in the behavior tree are stored internally and not as external and editable files in the Unity project. This can both be seen as an upside and downside, because an AI designer might want to reuse setting files in other trees without duplicating them, while on the other hand the project has fewer binary files to maintain and persist.

The graph editor UI and the accessibility to the most relevant functionalities, such as creating and maintain task nodes and data is implemented clearly, which results in a good user experience and maintainability. BD offers a large amount of action type tasks, which do simple operations, for instance fetching a primitive `C#` value from a component. This granular setup might be helpful to build up a more sophisticated structure, but also fills up too much space in the graph UI and in return equals rather a visual scripting tool, than an AIS in the first place. On the other hand, this can be avoided with custom built actions, which do all necessary operations to achieve a larger less granular goal.

As already detected BD uses the behavior tree decision-making algorithm as the main architecture and visual display. In Section 3.3 the requirements defined in Chapter 2 are discussed with the conclusion, that the behavior tree pattern meets the requirements for decision-making but not action management or data-injection and in addition partly has minor issues with the maintainability in regard to the node amount using condition nodes.

For data-management BD uses a custom-built architecture to dynamically add an inheritable object called *SharedVariable*. Most primitive C# types, as well as the most relevant Unity types are already prepared as *SharedVariables* by BD, for example the *SharedString* representing the string type. Said *SharedVariables* offer different possibilities to read the desired object type from, such as a constant or assigned value from another source.

All for the AI behavior necessary *SharedVariables* can be added, removed and maintained in a designated list for the relevant behavior tree. Assuming the *SharedVariables* are cached and don't have to be found by name (or hash) each time they are requested, it is a valid implementation to support custom data in amount and types. In that regard it can be compared to the blackboard architecture described in Section 3.7, although it is not injected in the actions as a parameter but rather used as a locally stored reference in a task.

Evaluation Process: Step 4

Creating custom data containers, actions and conditionals as per step 4 of the evaluation process works flawlessly from the AI programmer's perspective, due to the provided template files and automatic code generation features. After compilation the class types appear in the graph editor UI as a (node) creation option.

To setup the test scenario as described in step 4.e, the behavior tree graph including the custom action and condition tasks is built, as pictured in Figure 4-2 (excerpt) and Figure 4-3 (full overview).

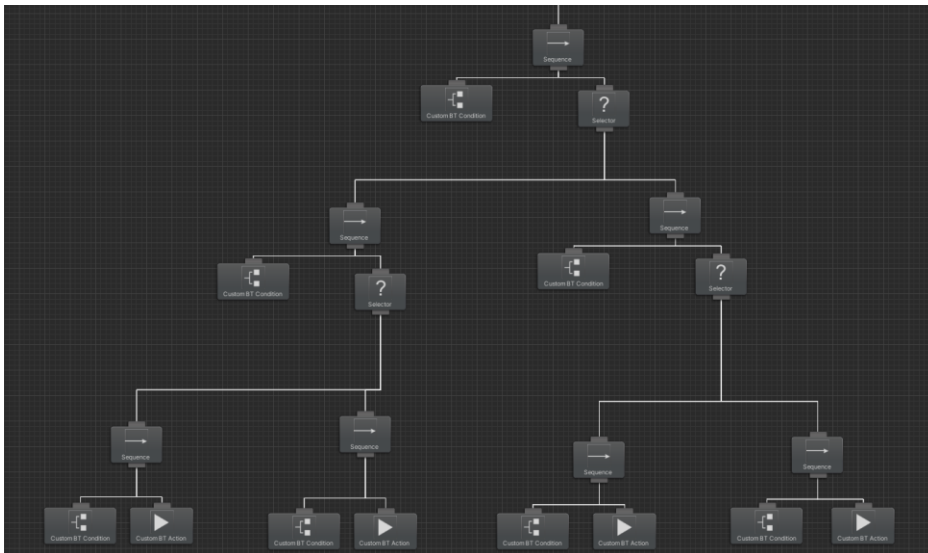


Figure 4-2: BD Benchmark Testing Graph Fragment

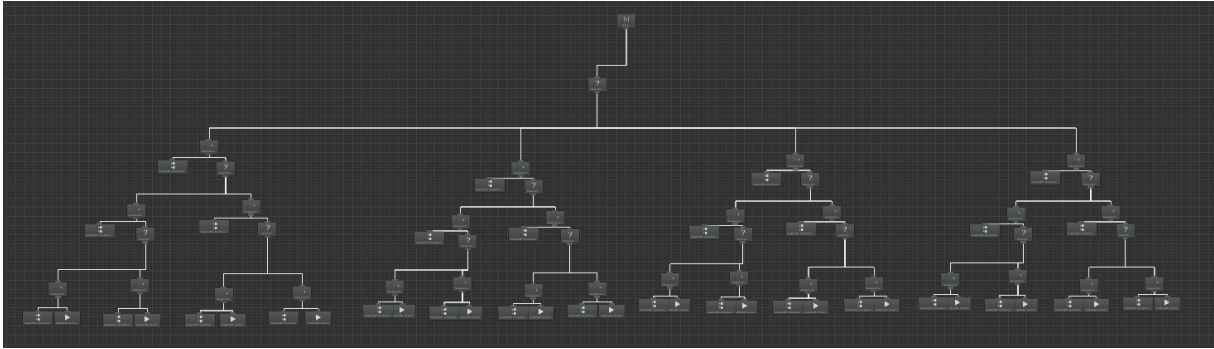


Figure 4-3: BD Benchmark Testing Graph Entire View

While building the graph, it is apparent that connecting nodes separated at a greater distance via the dragging connection functionality is tedious. A feature, such as connecting a node via direct reference (name or id) or via a picking option from a list of potential nodes available in the graph might solve this issue. Furthermore, the editing of multiple nodes of same type at once is prohibited, which is unfortunate regarding the AI setup time and general usability.

The graph Figure 4-4 displays the action, decision and total execution amount for each test case interval defined in step 4.f. It is apparent, that the decision calls dominate the action calls, since the behavior tree always must be traversed to execute an action and therefore add unnecessary overhead and waste performance, as stated in Section 3.3. Furthermore, it is visible that the total calls stagnate relative to the **by the factor 10** increasing agent amounts.

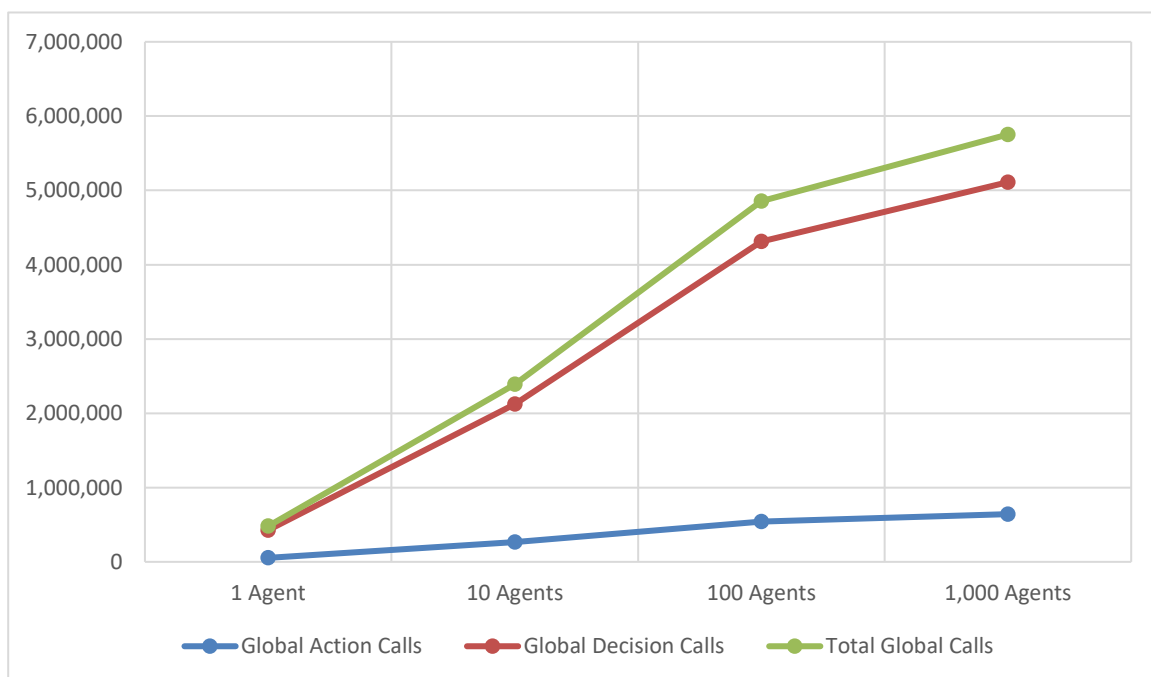


Figure 4-4: BD Benchmark Method Calls for 30s

Figure 4-5 visualizes the median CPU time usage of all called (AI) scripts per frame. The growth of the script execution time grows nearly linear to the increasing number of agents, considering the **agent amount** for each interval is **multiplied by 10**. For the extreme case of 1,000 agents calculated at once, the median script execution time per frame is roughly 30ms, which would lead to a stuttering gameplay experience in combination with other game relevant renderings and calculations.

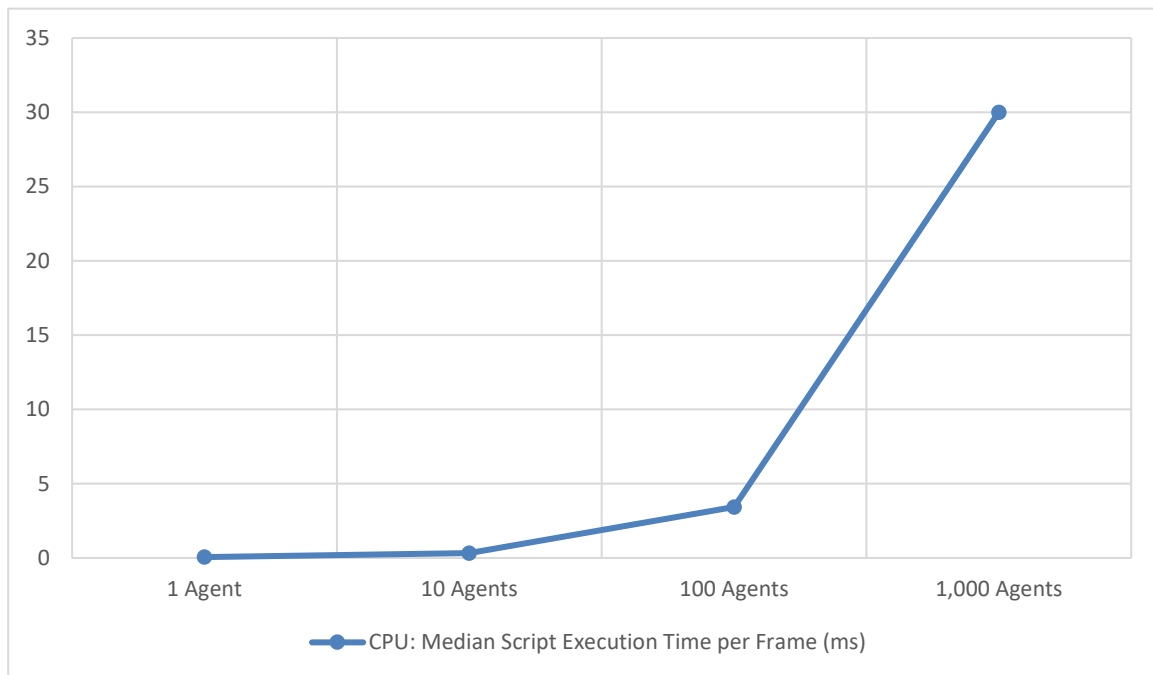


Figure 4-5: BD Benchmark CPU Usage

Table 4-2 illustrates the memory usage provided by the Unity3D Profiler, growing about linear relative to the increasing agent amount (**factor 10**) but not relative to the total calls amount displayed in Figure 4-4. Based on this, it is assumed that the increasing memory is rather tied to the GameObject count, than the executing scripts for this AIS.

Unlike the **GCAF** category, **GCUM** highlights a stair like growth, especially visible when calculating 10 to 1,000 agents at once, as it can be seen in Figure 4-6. Looking at the category **OC**, it is evident that the Unity3D Engine creates a base overhead of ca. 5,700 native objects and that a single added agent GameObject roughly represents 3-4 native objects based on the data presented.

	Memory: GCAF	Memory: GCUM	Memory: OC
1 Agent	1.7KB	16.1MB	5,710
10 Agents	11.4KB	15.5MB – 17.6MB	5,740
100 Agents	110KB	23MB – 26.9MB	6,100
1,000 Agents	1MB	110.1MB – 130.1MB	9,700

Table 4-2: BD Benchmark Memory Usage per Frame

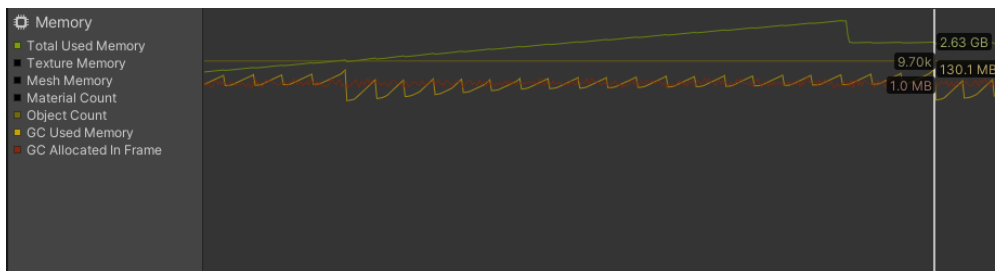


Figure 4-6: BD Memory Profiler Excerpt (1,000 Agents)

Summarizing the benchmark test results, this asset pack provides a well implemented memory management system, since the memory usage grows linear to the increasing agent amount and no significant memory leaks can be observed. As for the script execution time, the caused overhead by the decision-making algorithms proves the ineffectiveness of the decision tree and behavior tree pattern in regard to the action execution and therefore does **not meet the performance requirements** of Section 2.2. Furthermore, the AI developer must proceed with caution when building a Scene calculating lots of agents at once to prevent a low frames per second (FPS) rate. Other than that, the **scalability** and **maintainability** as of Section 2.1 and 2.3 are **provided** by this AIS.

4.3 [REDACTED]

Evaluation Process: Step 1

The first step, integrating the AIS [REDACTED] (EA3) version 3.0.0 as mentioned in the evaluation process step 1 causes no issues and the example Scenes are working as described.

Evaluation Process: Step 2 & 3

The minimum required Unity version by this AIS is 2018.4.27 and with that almost covers the minimum supported Unity version of the Asset Store. Furthermore, no excluded OSs can be found in the documentation [28] and store page [16].

Based on the documentation and sample Scenes it is not possible to detect a designated AI pattern, other than it might be an RBS as mentioned in Section 3.4. It is not clear how custom actions, conditions and data containers can be implemented into or expanded from this asset pack.

The AI behavior settings for an AI agent are located within one MonoBehaviour class as it can be seen in Figure 4-7. Special MonoBehaviour scripts are the only interface for the AI designer to maintain the AI behaviors in, thus lacking a tool to overview the AI behavior decisions and actions visually. Furthermore, the behavior data and behavior implementation are strictly defined within the core system and not built in a modular, but hardcoded manner and therefore providing an unsatisfactory code architecture [13].

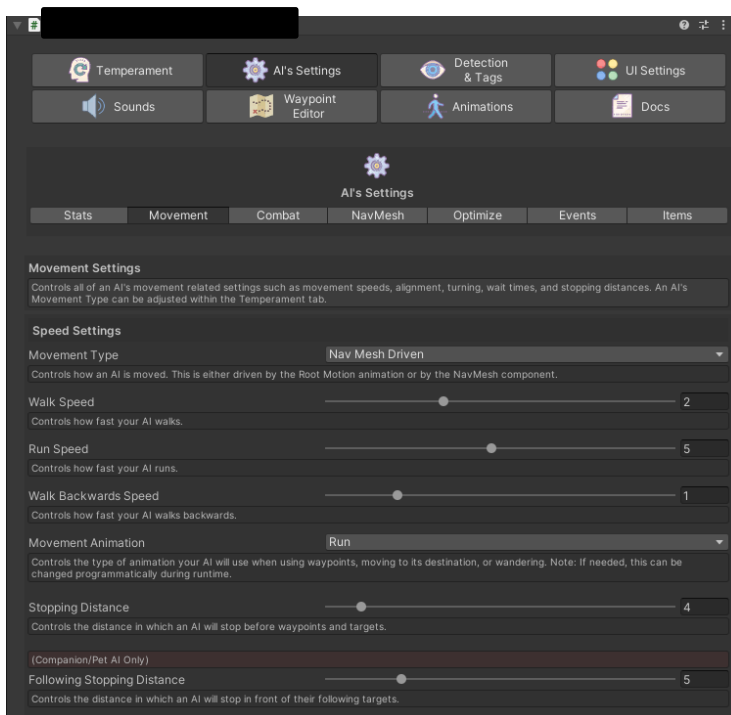


Figure 4-7: EA3 AI Behavior Setup

Inspecting the code as shown in Listing 4-3 reveals the assumption of this AIS being an RBS to be true and as discussed in Section 3.4 does not meet the requirements for a generic AIS (see Chapter 2).

```
961         if (UseDeactivateDelayRef == YesOrNo.Yes && DeactivateTimer >= DeactivateDelay |
962             {
963             Deactivate();
964             }
965         }
966         else if (!Renderer1.isVisible && !Renderer2.isVisible && !Renderer3.isVisible && Tot
967             {
968             DeactivateTimer += Time.deltaTime;
969
970             if (UseDeactivateDelayRef == YesOrNo.Yes && DeactivateTimer >= DeactivateDelay |
971                 {
972                 Deactivate();
973                 }
974             }
975         else if (!Renderer1.isVisible && !Renderer2.isVisible && !Renderer3.isVisible && !Ren
976             {
977             DeactivateTimer += Time.deltaTime;
978
979             if (UseDeactivateDelayRef == YesOrNo.Yes && DeactivateTimer >= DeactivateDelay |
980                 {
981                 Deactivate();
982                 }
983             }
984         }
985         else if (OptimizedStateRef == OptimizedState.Active)
986         {
987             if (TotalLODsRef == TotalLODsEnum.Two)
988             {
989                 if (Renderer1.isVisible || Renderer2.isVisible)
```

Listing 4-3: EA3 RBS Code Snippet

Evaluation Process: Step 4

Since this AIS does not provide expandability, it is not possible to create the test cases described in the evaluation process step 4. For that reason, this AIS does **not pass** the test cases, nor the **requirements** from Chapter 2.

4.4 [REDACTED]

Evaluation Process: Step 1

The integration of [REDACTED] (RVSAI) version 1.5 in the test project and the sample Scenes are working as described. Based on the store page [17] the minimum required Unity version is 2018.3.14 and no excluded OS platforms are to be found in the documentation [29].

Evaluation Process: Step 2 & 3

After creating a simple testing behavior as illustrated in Figure 4-9, it is apparent that this AIS implements a (binary) decision tree pattern, which is examined in Section 3.2. Unlike how it is stated in the manual to be based on the behavior tree pattern. Examining the manual excerpt Figure 4-8, as well as the test graph from Figure 4-9 indicates, that the tree structure used in this AIS combines the branch and leaf nodes to one shared node.

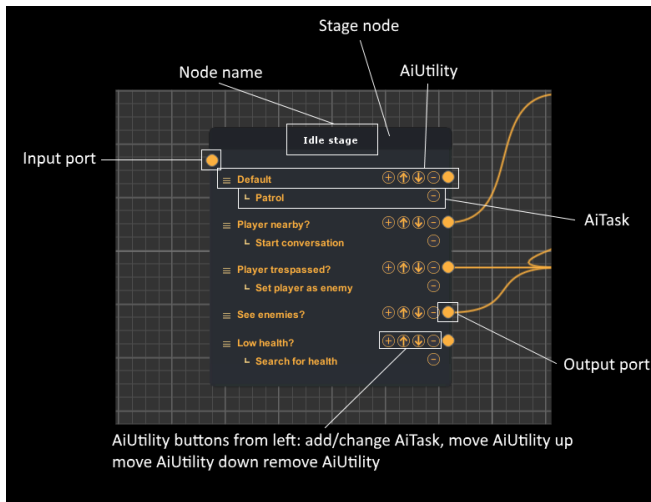


Figure 4-8: RVSAI Node Explanation [29]

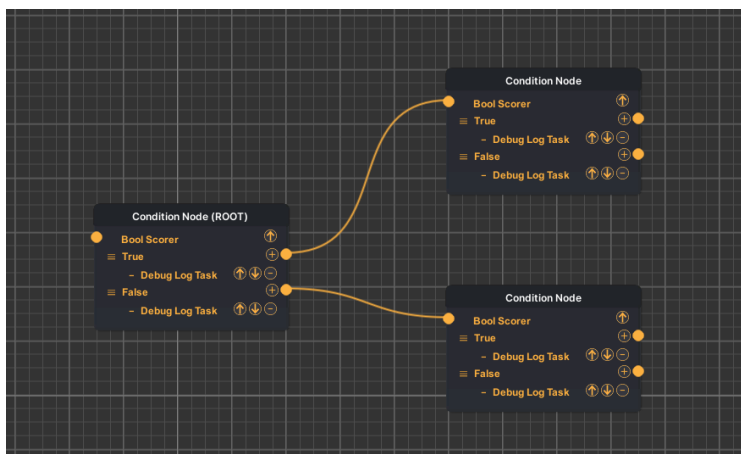


Figure 4-9: RVSAI Test AI Setup

The test graph as shown in Figure 4-9 results in a persistent Prefab file with the structure displayed in Figure 4-10. Since the graph and its nodes visualize settings for AI behavior and not instantiation of runtime GameObjects, it would have been more efficient to use ScriptableObjects instead for consistency and memory reasons, due to less overhead being generated by ScriptableObjects [4].

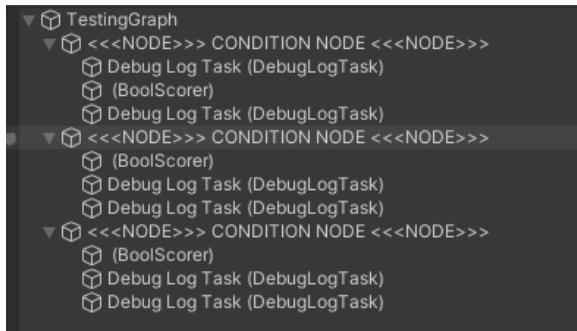


Figure 4-10: RVSAI Test Graph Prefab

As already discussed in Section 3.2, this AI pattern meets requirements for maintainability and performance regarding the decision-making procedure but not the action handling and data management. This AIS handles the action execution similar to the classic decision tree pattern and thus has to traverse the tree each time to execute an action.

RVSAI uses graph variables for global data similar to the (global) blackboard pattern described in Section 3.7 and is stored as a MonoBehaviour in the graph Prefab, visualized in Figure 4-11. As already discussed in Section 3.7, it is recommended to build a specific (local) blackboard for modularity and scalability reasons.

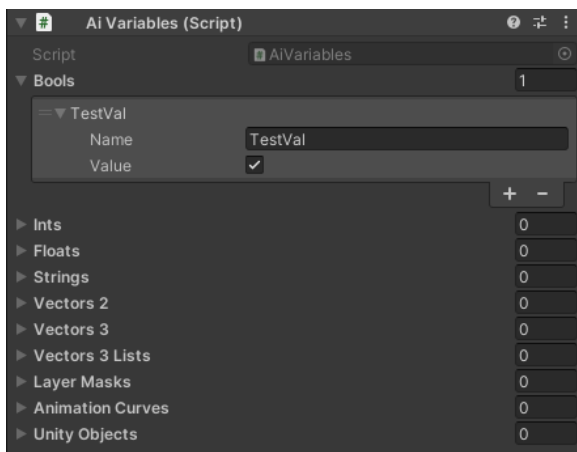


Figure 4-11: RVSAI Global Variables

Upon investigating the code, where and how the variables are requested, it is revealed that the required list item always must be fetched via its description name (see Listing 4-4). This is especially critical for performance reasons. For example, if 100 bool variables are stored within the same list, they must be fetched by name (string comparison) every time, at worst every frame and after n (in this example 100) comparisons.

```

8      public class VariableBoolProvider : BoolProvider
9      {
10     {
11         Fields
12     }
13     }
14     }
15     }
16     }
17     #region Not public methods
18     }
19     protected override bool ProvideData() => aiGraph.GraphAiVariables.GetBool(boolName);
20     }
21     #endregion
22     }

```

Listing 4-4: RVSAI Variable Bool Provider

For runtime and instance-based data RVSAI uses a context provider (see Listing 4-5), which can be inherited and specified for custom needs and is similar to the specified and more efficient (local) blackboard pattern.

```

107     /// <summary>
108     /// Shortcut for casting context to desired type
109     /// </summary>
110     protected T ContextAs<T>() where T : class => Context as T;

```

Listing 4-5: RVSAI Type Conversion

Evaluation Process: Step 4

RVSAI offers different approaches for AI decision-making and action execution with load balancing. While load balancing is active, the execution calls are limited to guarantee a stable and high FPS rate in order to prevent game stuttering. This configuration setup is a well-designed feature by RVSAI if lots of less important AI agents need to be calculated in the same Scene, while keeping the overall game experience as immersive and stable as possible. This setup (see Figure 4-12) is included in the benchmark tests.

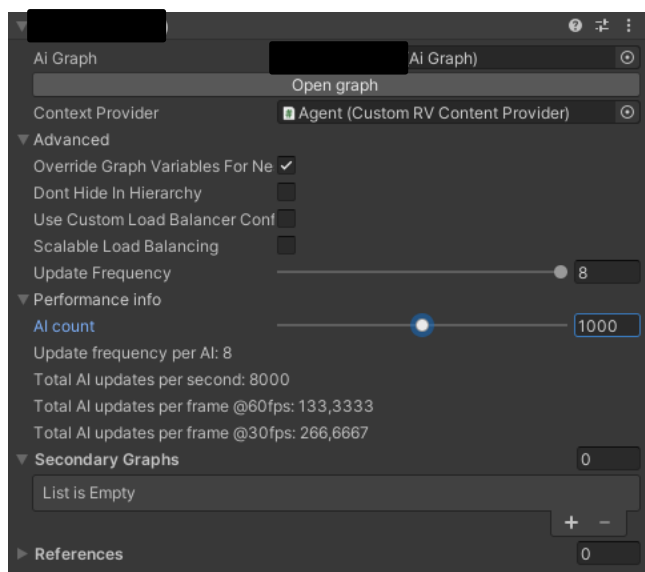


Figure 4-12: RVSAI Load Balanced AI Setup

Since the execution calls are reduced drastically during the load balancing setup, another configuration setup without RVSAI load balancing is included for the benchmark test, as shown in Figure 4-13.

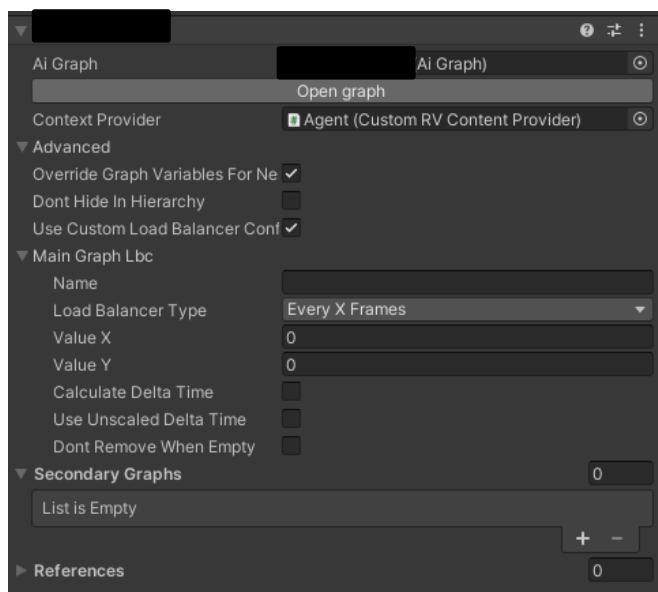


Figure 4-13: RVSAI Non-Load Balanced AI Setup

The manual explains that expandability for custom data and logic is granted for the AI developer with the help of script templates accessible via the context menu in the Unity project view [29].

Figure 4-14 represents the balanced decision tree using the custom scripts created for the benchmark test as required by step 4 of the evaluation process.

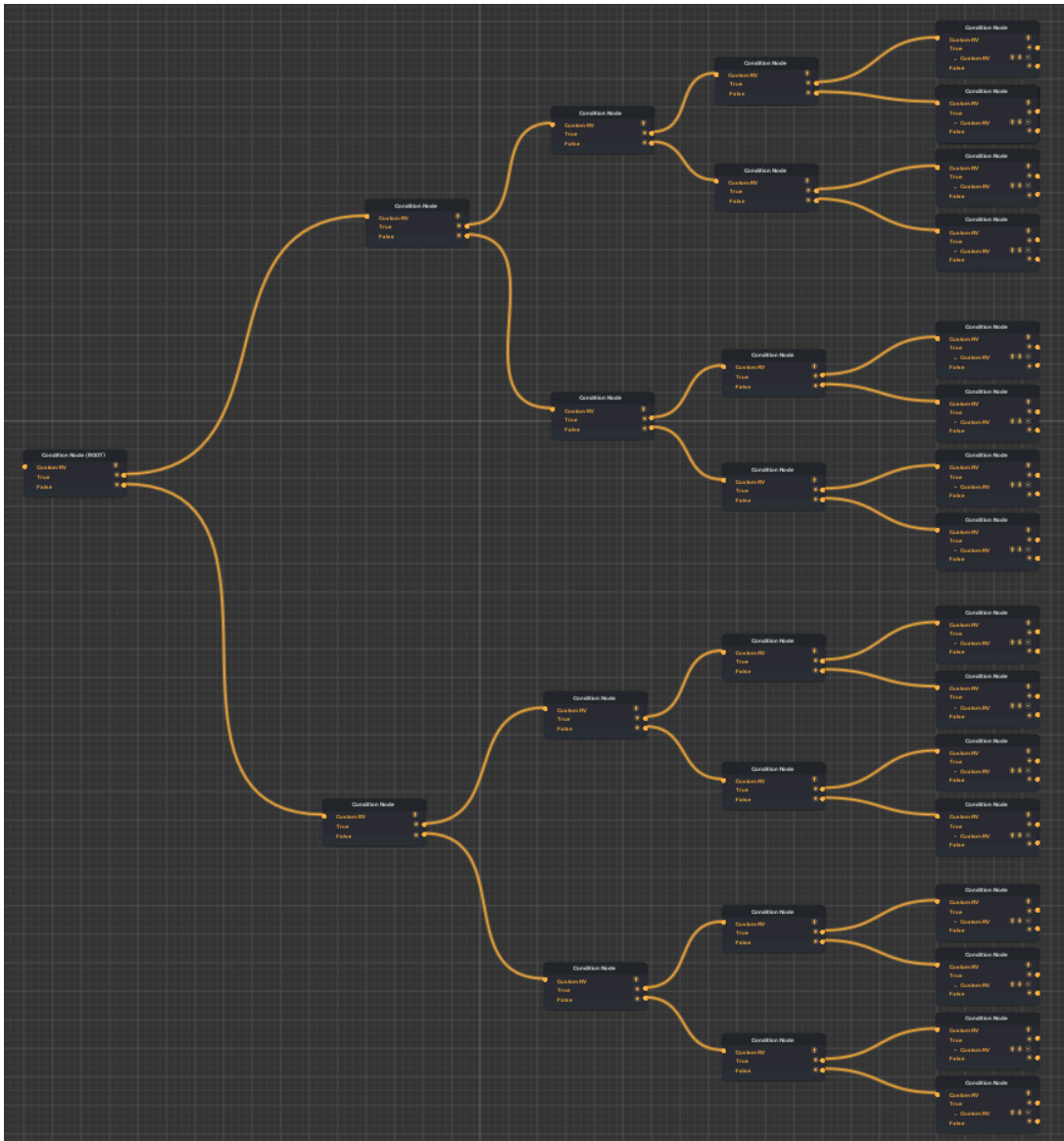


Figure 4-14: RVSAI Benchmark Test AI Behavior

The first benchmark tests use the non-load balancing setup from Figure 4-13 to examine how this asset pack behaves when maximum output (executions) is the highest priority.

Since it is not possible for the classic decision tree pattern to separate the decision-making system from the action execution, as stated in Section 3.2, the decision calls are significantly higher than the action calls (see Figure 4-15) and thus performance is wasted for redundant calls. Although the growth of total calls per agents does appear to be linear in this graph visually, it is not the case statistically due to the multiplication of agents **by the factor 10** for each new test interval (see step 4.f of the evaluation process).

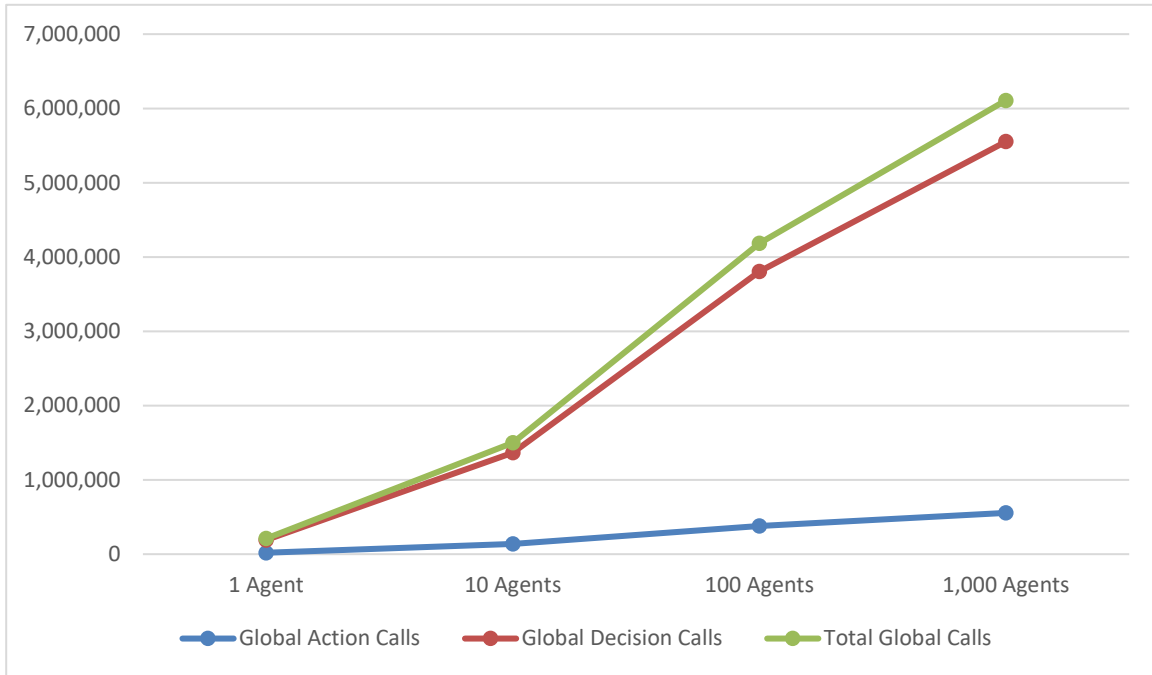


Figure 4-15: RVS AI Benchmark Method Calls for 30s (Non-Load Balanced)

Figure 4-16 showcases, that the median script execution time in ms per frame expands linear relative to the increasing agent amount **by the factor 10**, with the highest execution time being about 14ms enabling a smooth gameplay experience even in an extreme case of 1,000 calculated agents at once.

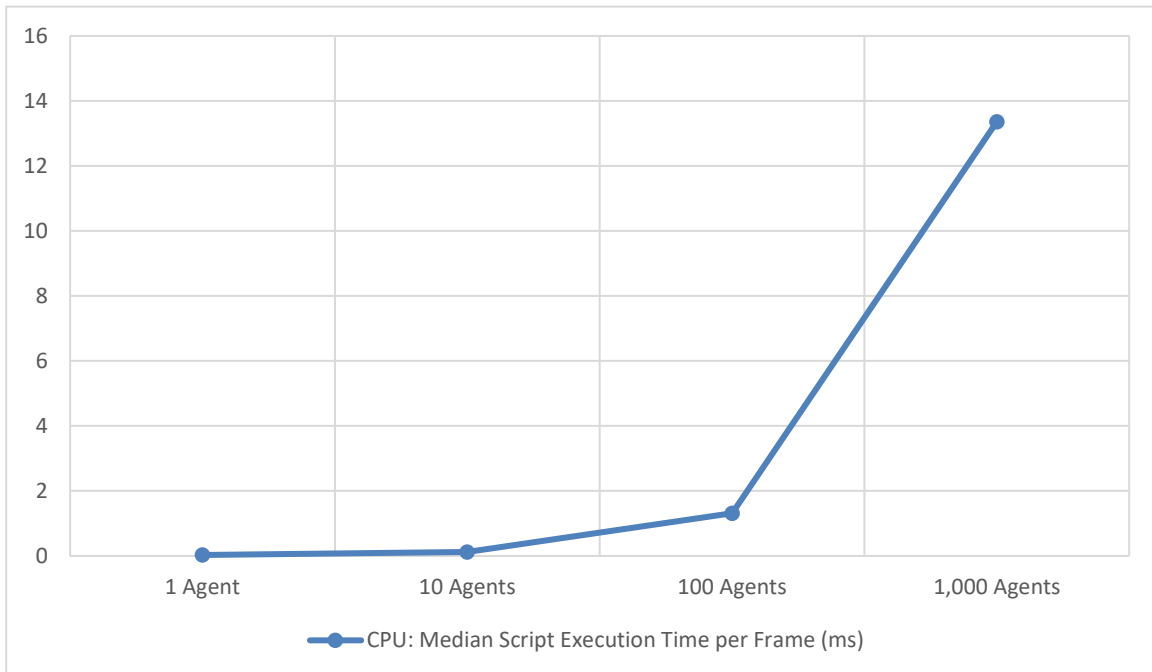


Figure 4-16: RVS AI Benchmark CPU Usage (Non-Load Balanced)

Similar to the memory Profiler results in Section 4.2, a stair like pattern can be observed showcasing how the GC is handled by this AIS and the Unity3D Engine. Furthermore, it is evident that the values of the categories **GCUM** and **GCAF** roughly grow linear in relation to the (times 10) increasing agent amount (see Table 4-3). What stands out, is the fact that the **OC** category of the memory module is exceptionally high. This is most likely due to the usage of Prefabs in combination with Components, instead of ScriptableObjects for the AI behavior settings. Using this built testing AI behavior results in 1 agent carrying roughly 270 native objects, indicating that the more settings the AI behavior graph contains, the more memory is used in an inefficient manner.

	Memory: GCAF	Memory: GCUM	Memory: OC
1 Agent	284B – 426B	17.3MB	6,280
10 Agents	3.7KB – 4.9KB	17.1MB – 19.2MB	8,640
100 Agents	40.1KB – 48.6KB	27MB – 30.2MB	32,130
1,000 Agents	388.4KB – 474KB	138MB – 150.2MB	267,030

Table 4-3: RVSAI Benchmark Memory Usage per Frame (Non-Load Balanced)

Using the load balanced setup as displayed in Figure 4-12, it is apparent that the total amount of calls grows linear relative to increasing amount of agents (see Figure 4-17), but are generally exceptionally less compared to Figure 4-15. Since the same architecture is used, the decision-calls dominate the action calls as well.

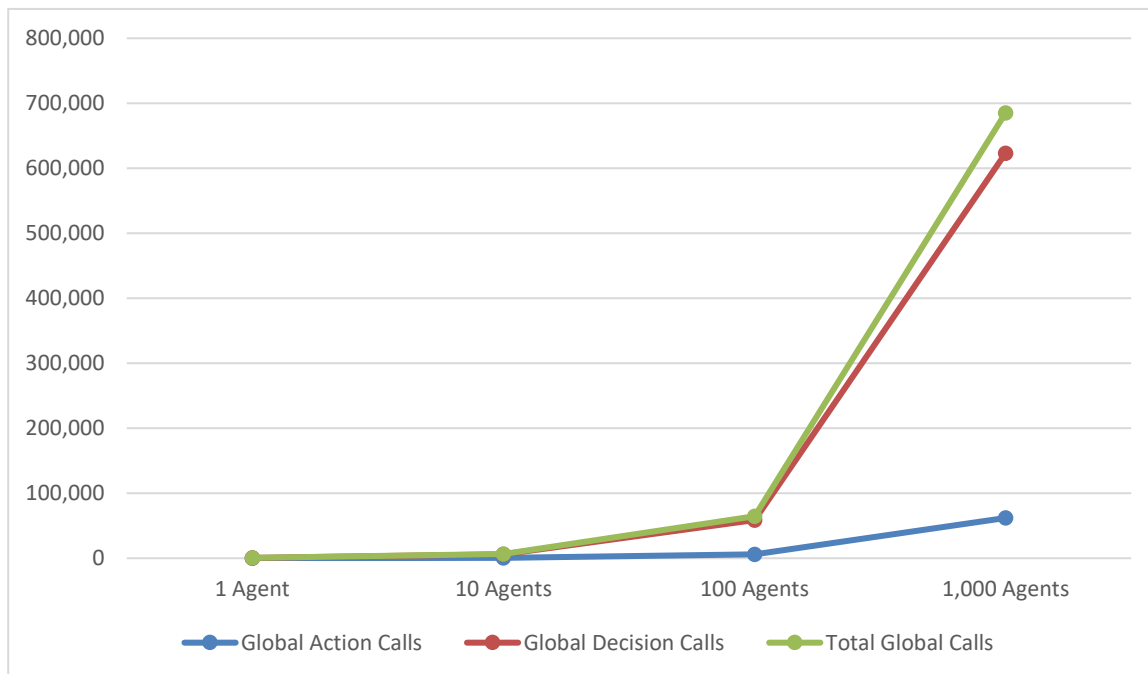


Figure 4-17: RVSAI Benchmark Method Calls for 30s (Load Balanced)

Figure 4-18 displays a constant median script execution time for an agent amount between 1 and 100 and increases by the factor 6 for 1,000 agents. A misleading representation due to the dominating overhead Unity generates for the first 100 agents. This concludes that the load balancing system provided by this asset pack efficiently stabilizes the FPS rate to ensure a smooth gameplay experience, with the cost of executing very little decisions and actions (ca. 5.5million calls less for 1,000 agents).

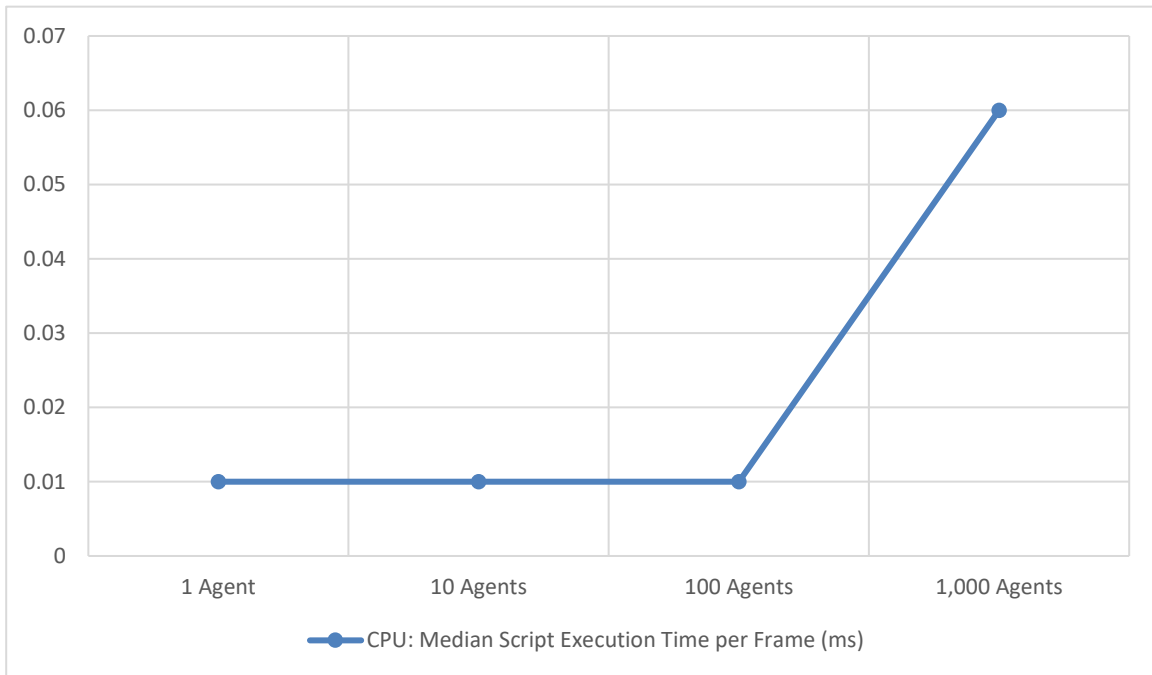


Figure 4-18: RVS AI Benchmark CPU Usage (Load Balanced)

Table 4-4 shares similarities to the observation of Table 4-3 in regard to the **OC** and the **GCUM** categories. The only exception being the category **GCAF** showcasing a drastically smaller growth and overall data usage most likely caused by less total decision and action calls.

	Memory: GCAF	Memory: GCUM	Memory: OC
1 Agent	0B – 390B	17.7MB	6,260
10 Agents	0B – 474B	22.1MB	8,650
100 Agents	0B – 390B – 474B	28.2MB – 32.1MB	32,140
1,000 Agents	316B – 1.2KB – 2.3KB	138.4MB	267,050

Table 4-4: RVS AI Benchmark Memory Usage per Frame (Load Balanced)

Concluding the benchmark tests, this asset pack provides efficient script execution times with or without the load balancing feature enabling stable FPS rates. The biggest issue being the memory management and potential danger when using large AI behavior trees, which is likely caused by using Prefabs as settings. Furthermore, the ineffectiveness of the decision tree pattern for executing actions is proven as discussed in Section 3.2 and therefore the **performance requirement** as per Section 2.2 is **not met**.

The **maintainability** of the graph is rather unusual but **provides** the core necessary features to build an AI behavior, as well as the **scalability**.

4.5 [REDACTED]

Evaluation Process: Step 1

Integrating [REDACTED] (FSMAIT) (version 1.1.7) into the test project and running its example Scenes works as described in the store page [18] and manual [30].

Evaluation Process: Step 2 & 3

Based on the store page and documentation this asset pack has no excluded OSs it can run on and the minimum compatible Unity version is 2019.4.22. Furthermore, it is revealed that this AIS implements the FSM pattern for decision-making as described in Section 3.1. The FSM approach is as already revealed not the most maintainable option regarding the growth of the graph elements (states and transitions) compared to the decision tree or behavior tree patterns (see Section 3.2 and 3.3). In addition, this asset pack forces decision-making and action execution each time the FSM is updated, although it is theoretically possible to customize the architecture of this AIS via inheritance.

The FSM graph is stored a ScriptableObject and the nodes (states) as well, although only the graph is accessible to the user as a binary file in the project. States can contain two types of components, both of which inherit from the ScriptableObject class but are not written into a separate binary file in the project either (contained within the graph ScriptableObject). The component types are either an action to execute or a decision for potential transitions to other states.

The first created FSM graph to test the by this AIS provided features is displayed in Figure 4-19 and simply logs the current visited state. Generation of custom actions and decisions can be done fairly simple via the context click option.

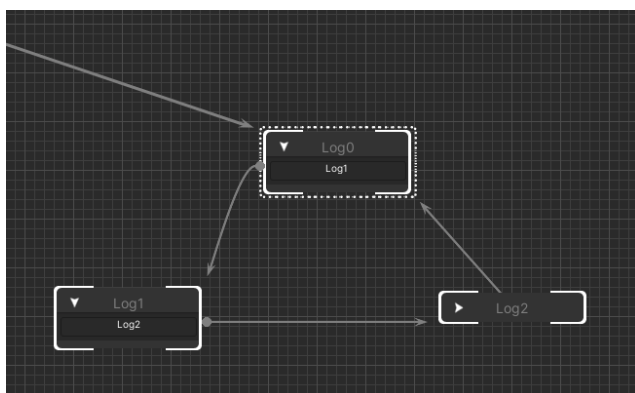


Figure 4-19: FSMAIT Graph Test

Attempting to create a new blank AI to run the test graph from Figure 4-19 leads to difficulties, one being a cropped UI window (see Figure 4-20) and the other being not necessary requirements, such as a rigged humanoid model and an animation controller for the requested humanoid model.

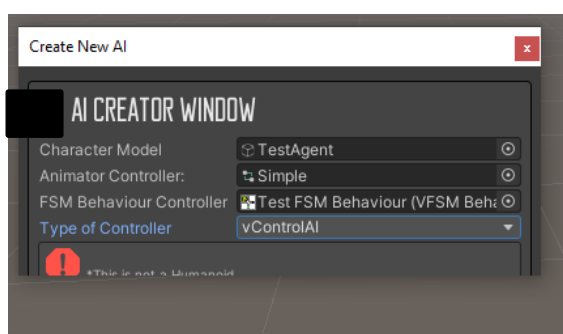


Figure 4-20: FSMAIT Editor Bug

If the AI creation setup is skipped and attempted manually without an animator controller and humanoid rig, the FSM either doesn't work or throws errors in the console indicating that required Components of the GameObject are missing, although the tested FSM as shown in Figure 4-19 only handles logging actions (not needing a humanoid rig or animator controller). In order to solve the redundant requirements issue, a custom FSM controller has to be written, which inherits from *vControlAI* provided by this asset pack. Implementing this interface leads to further issues, including the forced implementation of specific methods, such as *ResetHealth*, *LookAtTarget* and many more. It appears, that those methods are behavior scripts specific for humanoid AI, that should rather be refactored into separate actions to justify the generic AI architecture by FSMAIT but as of now are violating the single responsibility principle [13].

In order to get custom data to the (custom) actions and decisions, a new FSM behavior class with the relevant custom data is created. This class must inherit from *vIFSMBehaviorController*, due to it being the external runtime data container the actions and decisions get as a parameter (see line 25 of Listing 4-6). The specific required FSM behavior class then can be casted from this interface.

```
20 public override void DoAction(vIFSMBehaviourController fsmBehaviour,
21     vFSMComponentExecutionType executionType = vFSMComponentExecutionType.OnStateUpdate)
22 {
23     EvalHelper.WastePerformance();
24
25     CustomFSMController customFsmController = (CustomFSMController) fsmBehaviour.aiController;
26     CustomDataContainer dataContainer = customFsmController.customDataContainer;
27
28     dataContainer.actionsCounter++;
29     dataContainer.actionsName = customFsmController.name + " Action " + dataContainer.actionsCounter;
30
31     dataContainer.customSharedDataComponent.totalGlobalActionsCount++;
32
33     //Debug.Log(log);
34 }
```

Listing 4-6: FSMAIT Benchmark Custom Action

This solution equals the global blackboard pattern discussed in Section 3.7, since the *vIFSMBehaviorController* is tied directly to the FSM and action execution and is compared to the local blackboard pattern not the best implementation in terms of scalability and performance. Using the cast operator possibly each frame could be avoided if a specific type is declared as the parameter for the respective action classes.

Evaluation Process: Step 4

To benchmark the FSM according to the evaluation process defined in step 4.f, which states that every leaf must be reachable from the current AI state, the graph shown in Figure 4-21 is built. This design though does not represent a regular FSM setup and a second graph Figure 4-22 is created, assimilating a common FSM design and is tested separately.

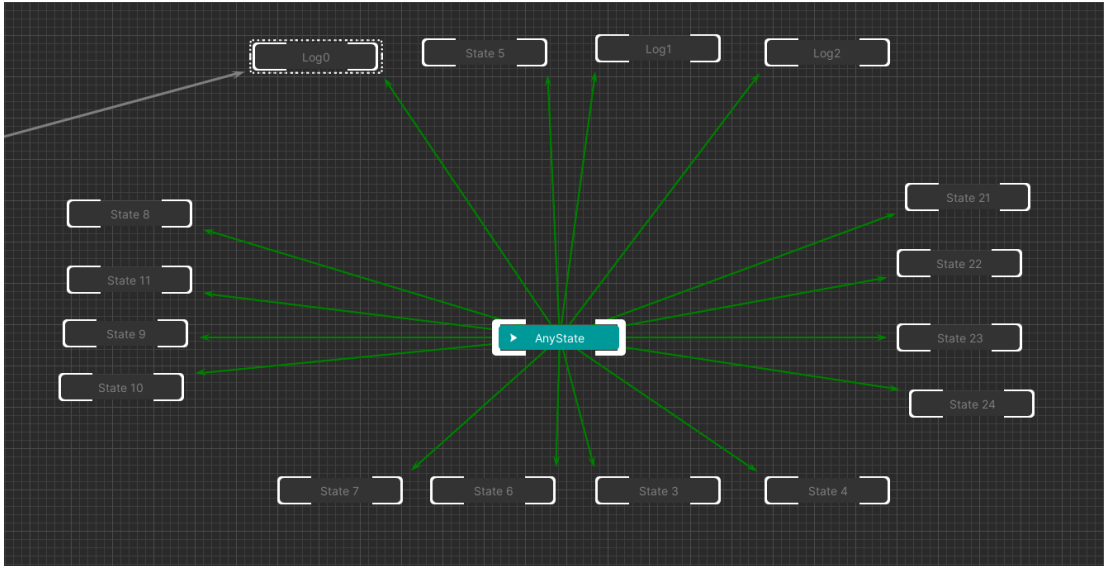


Figure 4-21: FSMAIT Benchmark Setup - Any State

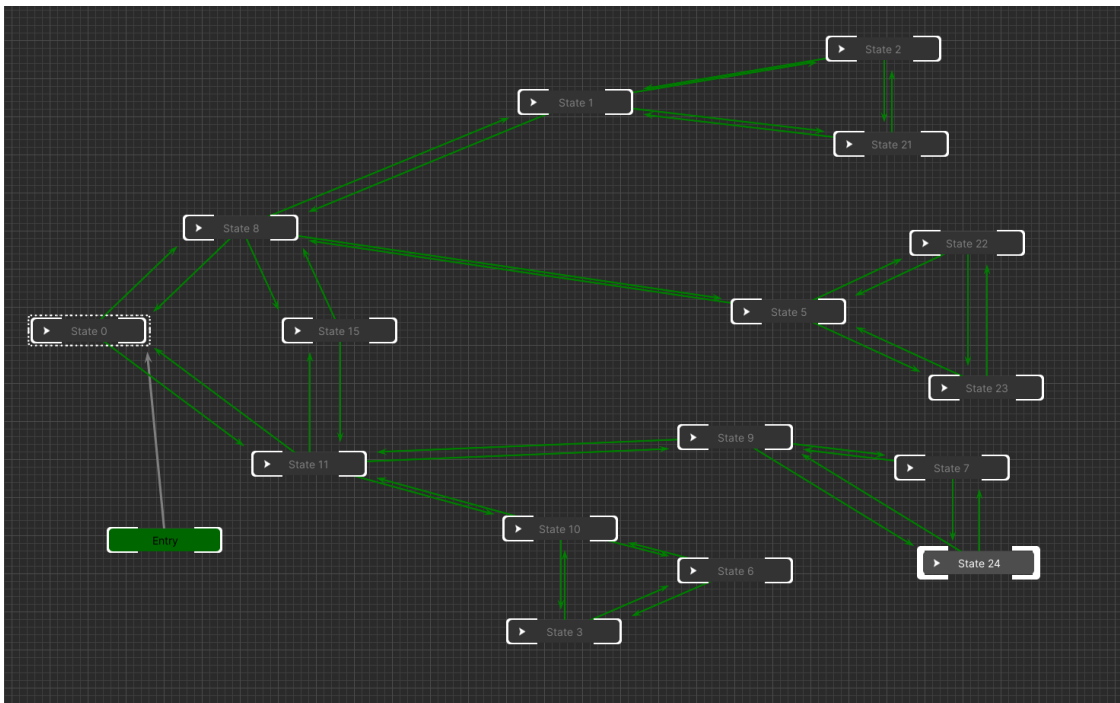


Figure 4-22: FSMAIT Benchmark Setup - Classic

Building a balanced decision tree from this FSM is not possible, since the FSM does not traverse to a destination node of type leaf, but rather treats every state as a leaf and potential final destination, which results in less actions being executed if a branch state (without an action) is the destination of the decision result.

While building the FSM graphs, following features were missing, providing an insufficient usability, user experience and maintainability:

- Duplicating and copying state nodes
- Zooming in and out, while working in the graph editor
- Editing multiple state nodes of same type at once

Figure 4-23 illustrates a stagnating curve of total execution calls with the decision calls dominating the action calls. This is due to more potential decisions to make and randomly fail, since each state has multiple transitions (with attached decision scripts) but only one action per state to execute, resulting in transitions/ decisions > states/ actions. Additionally, this asset pack does not provide separation of changing the state and updating a state's action and thus forcing the decision-making procedure before executing an action.

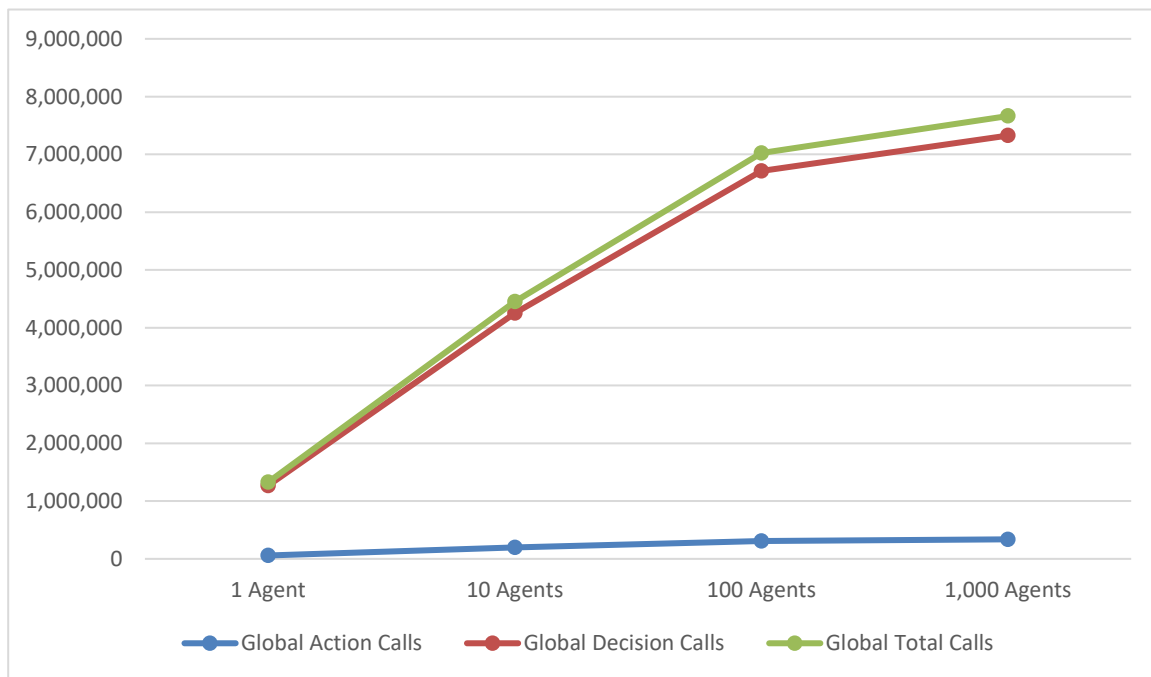


Figure 4-23: FSMAIT Benchmark Method Calls for 30s (Classic Setup)

As pictured in Figure 4-24 the ms rate roughly expands linear to the increasing number of agents with a critically high rate of ca. 80ms for 1,000 simultaneously calculated agents. In fact, the benchmark test with 1,000 agents at once does not capture the 30 seconds in 600 frames as shown in Figure 4-25.

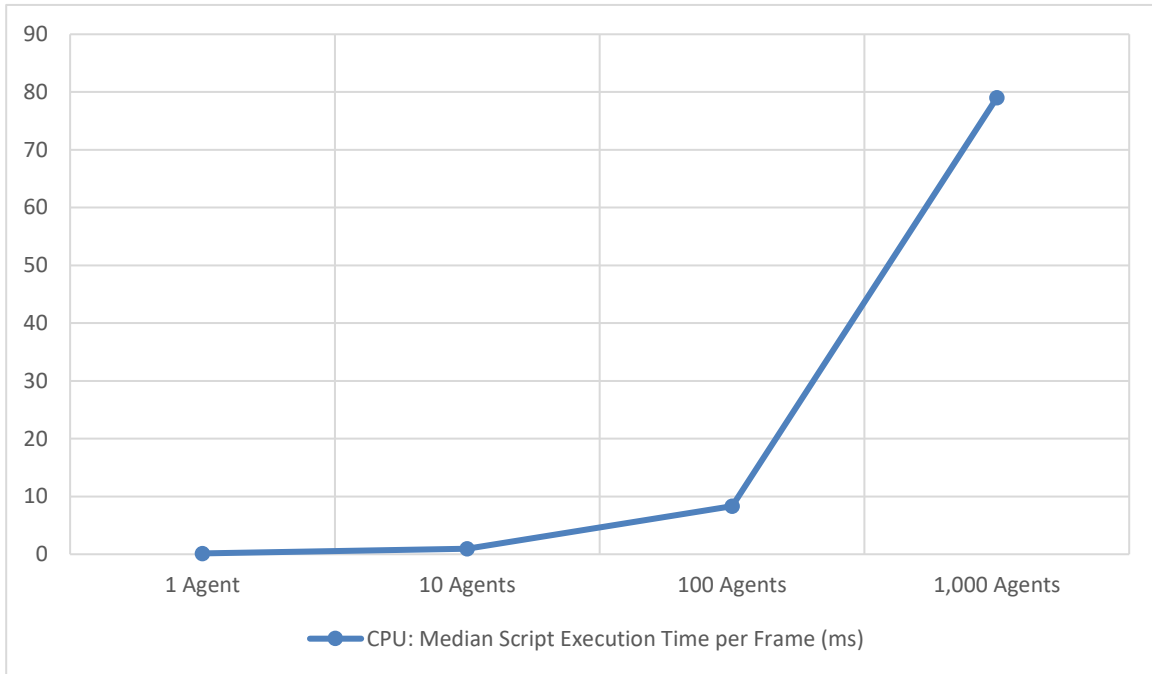


Figure 4-24: FSMAIT Benchmark CPU Usage (Classic Setup)

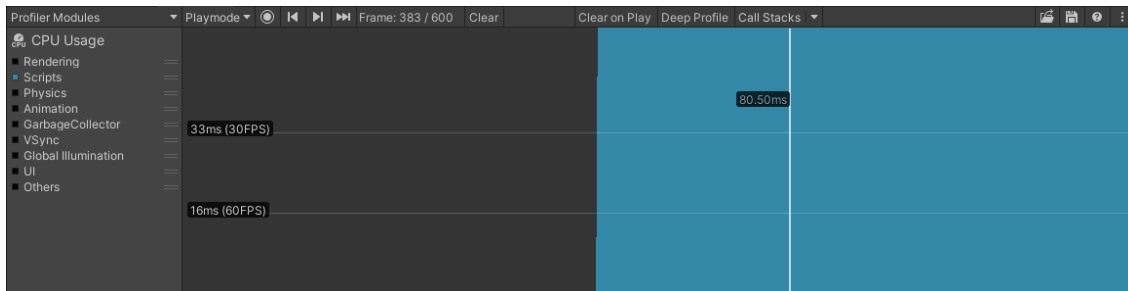


Figure 4-25: FSMAIT CPU Profiler Excerpt (1,000 Agents & Classic Setup)

The data provided by Table 4-5 indicates, that the **GCAF** category increases by the factor 10 for each row, which is linear to the increasing number of agents provided on the left-hand side of the table. The category **GCUM** showcases, that the growth is linear as well, although by a much smaller value per added agent. Both categories are fluctuating in their memory usage in each row, due to the internal GC activities in *C#* and Unity. Furthermore, it is apparent that each agent roughly corresponds to 4 native objects as it can be read and calculated from the column **OC**.

	Memory: GCAF	Memory: GCUM	Memory: OC
1 Agent	3.1KB – 3.4KB	14.8MB – 16.7MB	6,160
10 Agents	36.3KB – 37.2KB	14.1MB – 16.8MB	6,200
100 Agents	362KB – 365KB	16.4MB – 19.2MB	6,560
1,000 Agents	3.4MB – 3.5MB	24.9MB – 28.6MB	10,170

Table 4-5: FSMAIT Benchmark Memory Usage (Classic Setup)

Figure 4-26 displays the execution calls using the AI behavior graph displayed in Figure 4-21, with the decision calls overruling the action calls, although not as drastically as shown in Figure 4-23. This is due to less transitions and thus decisions being called relative to the existing states with their attached actions. Nevertheless, there are 16 potential (failing) transitions, each decision-cycle in relation to 1 executing action.

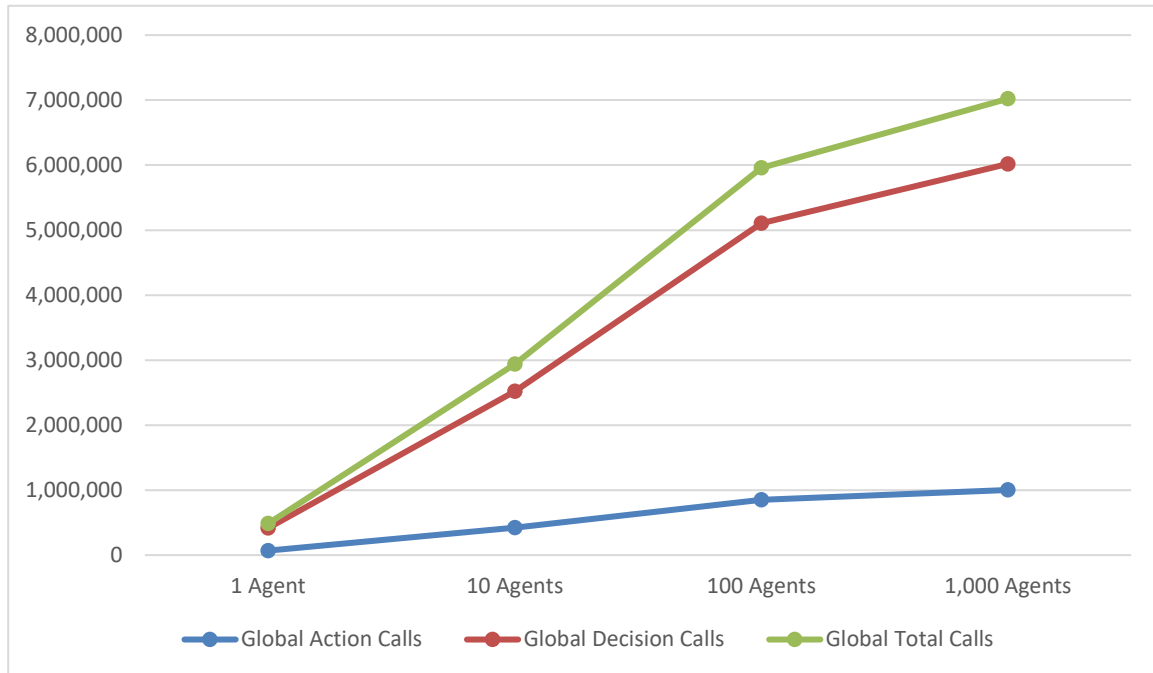


Figure 4-26: FSMAIT Benchmark Method Calls for 30s (Any State Setup)

Less critical median ms rates compared to Figure 4-24 with almost the same total execution calls (see Figure 4-23 and Figure 4-26), this Figure 4-27 shows a linear expansion of median ms rates per frame relative to the increasing number of agents, proving that the decision-making system is exceptionally more expensive than the action execution system.

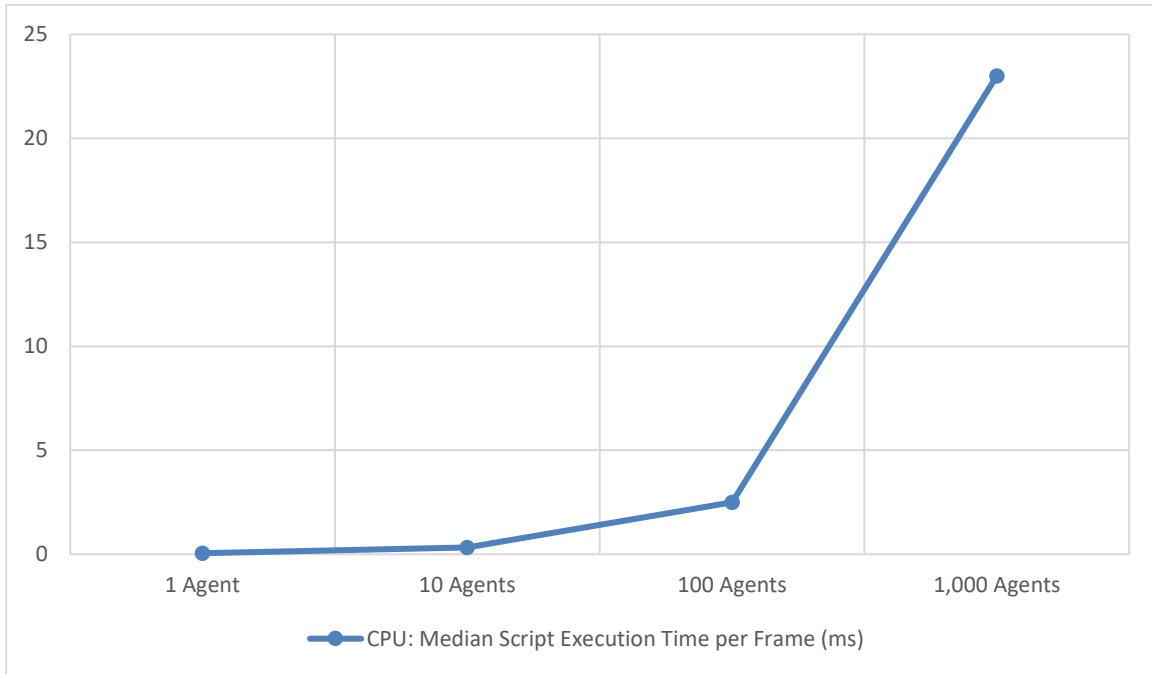


Figure 4-27: FSMAIT Benchmark CPU Usage (Any State Setup)

Both categories **GCAF** and **GCUM** displayed in Table 4-6 grow linear for each newly added agent, although there is an inconsistency for the category **GCUM** in the agent range between 10 and 100, most likely caused by the Unity3D Engine, otherwise not explainable. Each agent adds round about 4 native objects as it can be seen in the **OC** column, thus expanding linear relative to the agent count as well.

	Memory: GCAF	Memory: GCUM	Memory: OC
1 Agent	0.8KB – 1.7KB	15.7MB – 17MB	5,730
10 Agents	11KB – 13.6KB	16.6MB – 18.6MB	5,770
100 Agents	110KB – 117KB	14.6MB – 17.7MB	6,130
1,000 Agents	1.1MB	19.9MB – 26.7MB	9,730

Table 4-6: FSMAIT Benchmark Memory Usage per Frame (Any State Setup)

The benchmark tests of this AI asset pack concludes that the decision calls are exceptionally expensive, highlighting that the AI designer should proceed with caution when building an FSM AI behavior with lots of transitions and agents being simultaneously calculated in a single Scene. At best the AI programmer should implement a custom architecture that splits the decision-making system from the action execution system via inheritance, therefore the **performance requirement** from Section 2.2 is **not met** for this AIS. Other than that, the memory usage and management of this asset pack is efficient and economical.

Overall, this asset pack has the potential to provide **scalability** (see Section 2.1) but **fails** in execution of it, due to the redundant code base, although it is possible to work around it if the unused methods are ignored. As for the **maintainability**, the graph editing tool **lacks** crucial features to provide fast and efficient AI behavior creation and maintenance.

4.6 [REDACTED]

Evaluation Process: Step 1

The integration of the asset pack [REDACTED] (UAIS), version 1.7 works with no issues. Based on the documentation [31], as well as the store page [19] this AIS supports all OSs and requires at least the Unity version 2019.4.24 to run on.

Evaluation Process: Step 2 & 3

The provided sample Scenes showcasing different AI behavior, such as attack or follow player with different models seem to work upon first impression but have some issues either in path finding or decision making, due to the presented unexpected behavior.

The supplied documentation does not explain the decision-making pattern, as well as the custom action and condition creation for the AI developer. Looking into the code reveals, that this AIS implements the RBS pattern as discussed in Section 3.4 in a hardcoded and non-modular way (see Listing 4-7), unlike how this AIS is advertised in the asset store at the time this AIS is evaluated.

```
987     if (AttackOnGoing)
988     {
989
990         if (Vector3.Distance(transform.position, General.PlayerResources.PlayerObject.transform.position) >=
991             Animal.AnimalAttackSettings.MaxDamageDistance)
992         {
993             General.GeneralResources.Animator.SetBool(name:"Attack", value:false);
994             AttackOnGoing = false;
995             StartCoroutine(routine:AttackDone());
996         }
997     }
998
999     General.GeneralResources.Animator.SetBool(name:"Combating",PlayerDetected);
1000     if (!PlayerDetected && LastKnownPlayerPos == Vector3.zero && !Flee)
1001     {
1002         AnimalWander();
1003     }
1004
1005     else if(LastKnownPlayerPos == Vector3.zero && Animal.AnimalProperties.AnimalState == AnimalState.Brave &&
1006     {
1007         ChasePlayer();
1008     }
1009     else if(Animal.AnimalProperties.AnimalState == AnimalState.Coward && LastKnownPlayerPos == Vector3.zero &&
1010     {
1011         SetNavmesh(stop:true);
1012         flee();
1013     }
```

Listing 4-7: UAIS RBS Code Snippet

Evaluation Process: Step 4

As a result, this AIS does **not provide** the **requirements** defined in Chapter 2 and the benchmark tests as described in the evaluation process are not achievable, due to the lacking expandability and modularity of the core architecture.

4.7 Conclusion

Some asset packs available at the Unity Asset Store (see Section 4.3 and 4.6) do not offer an expandable architecture as advertised and therefore cannot be tested and validated to be generic AISs.

Those AI packs, which do validate the benchmark test defined in step 4 of Section 4.1 share the same issue that the decision-making system is convoluted with the action execution, always wasting performance for use cases that do not require decision-making calls each frame. With that observation, the theory stated in the AI patterns evaluation (see Section 3.10) is proven and that a combination of multiple (modified) frameworks is a necessity to meet the requirements defined in Chapter 2. Furthermore, the benchmark test results showcase linear growth in memory and CPU time usage per added agent, indicating that the implemented AI patterns cause no memory leaks or inefficient data handling, except for the evaluated AI pack in Section 4.4.

Using script generation tools based on template files, as it is provided by each validated asset pack (in terms of testing and scalability) is crucial for fast and efficient custom code creation while working with a generic AIS. As for the maintainability, most evaluated AISs provide a graph editor to visually display a custom setup AI behavior with its potential decisions and actions to execute, hinting that a graph editor is mandatory for a generic AIS to build large and complex AI behaviors.

5 Prototype Development

The development of the AIS prototype is subject of this chapter, considering the findings of the researched and evaluated AI patterns in Chapter 3 and the AI asset packs in Chapter 4.

First of all, to ensure a stable and organized development environment, the project is set up with a version control system, specifically git [Tool-3]. For task organization the issue board system of GitLab [Tool-4] is used, which additionally hosts the projects git repository enabling the feature to link commits to tasks and display them visually in the GitLab issue board.

The Unity version containing this project is the from the Asset Store minimum supported Unity version 2018.4.0f1, to potentially reach as much Unity developers as possible.

As concluded in Section 3.10 and 4.7, a combination of patterns must be used to ensure a scalable, maintainable and performant AIS and separately cover tasks, such as:

- Decision-making
- Action management
- Data management

For decision-making either the behavior tree from Section 3.3 or the decision tree as discussed in Section 3.2 are the best patterns to make use of. While the behavior tree offers a good architecture for expandability by providing the possibility to inherit from tasks, such as *composites*, *actions*, *conditionals* and *decorators*, it has maintainability issues regarding the node amount, causing the tree to potentially grow unnecessarily big. The decision tree on the other hand offers a more simplified version of a tree structure with a clean separation of branch and leaf nodes but lacks options for more sophisticated decision-making. Which tree pattern is chosen and how it is modified to solve its issues is further discussed in Section 5.1.4.

For action management the scheduler pattern (see Section 3.9) is the most promising choice, although there might be some difficulties defining execution time per frame for an action using the Unity3D Engine, since synchronous method calls using the native Unity API cannot be interrupted. Keeping this in mind, as well as the limited development time for this AIS prototype, the built scheduler does not include all features a regular scheduler would and is further highlighted in Section 5.3.

Splitting action management and decision-making into two separate modules with their own life cycles and execution times adds the question in how to know when and which previously

chosen action should stop in the next decision cycle? The solution to this uncertainty is at first established in Section 5.1.1, 5.1.3 and answered in Section 5.3.

For data management the blackboard pattern in its local form as investigated in Section 3.7 is the only valid option for this AIS prototype and is introduced in Section 5.3, as well as in Section 5.4.

Taking the evaluated asset packs from Chapter 4 into account, the best maintenance and usability is provided by BD and its node graph editor tool. Its advantages and disadvantages, as well as the ones from the other AISs are included in Section 5.7.

To increase the readability and maintainability of classes displayed in the Unity Inspector the modified open-source project NaughtyAttributes [Extern-1] is included in this AIS prototype, which provides C# attributes for the programmer that alter the visual output of objects rendered in the Unity Inspector.

In order to maintain the projects (custom) AI behavior, some form of configuration architecture must exist within this prototype, providing accessibility, as well as expandability for the AI designer and programmer. This architecture is further described in the following Section 5.1.

5.1 Settings

To provide a positive usability and user experience for the AI designer maintaining his or her custom AI behavior using this AIS prototype, the AI settings are accessible and editable via a GUI, specifically by the Unity Inspector and therefore not in a (pre-) compiled script file.

Two base approaches making use of Unity objects and the Inspector are feasible:

- Components
- ScriptableObjects

Using Unity's Component system requires a GameObject as a host and is generally only accessible as an instance in a Scene [2]. The exception to this is the usage of Unity's Prefab system, which enables the developer to persist a GameObject as a template in the Unity project without being dependent of a specific Scene [3]. It is revealed in comparison with the asset pack RVSAI outlined in Section 4.4 and the other evaluated and validated asset packs that using ScriptableObjects instead of Prefabs for AI settings is recommended to save memory overhead. Furthermore, ScriptableObjects are intended to represent settings files in the first place, unlike Prefabs which are intended to be instantiated in a Unity Scene and potentially interact or be interacted with the game environment [4].

The evaluated asset packs FSMAIT (see Section 4.5) and BD (see Section 4.2) both use ScriptableObjects as persistent files for the AI graphs only but not its nodes. This approach has advantages in terms of a more lightweight and cleaner Unity project but forces the AI developer to potentially recreate settings, which could otherwise be reused in other similar AI behavior graph. Furthermore, this might be the root issue of why the multi-editing feature is not supported by both AISs. For those reasons, setting files (ScriptableObjects) created in this AIS prototype are always persisted and accessible in the Unity project individually.

Considering the chosen AI patterns for this AIS prototype, following base setting types are required:

- State Settings
- Condition Settings
- Action Settings
- Decider Settings

Each setting type inherits from the *OneAIBaseNotesScriptableObject* class, adding optional Unity Editor only comment and naming fields for the AI designer to further improve the organization and maintainability of an AI project using this prototype.

Since the decision-making and action management tasks are separated from each other, a form of identification is required for actions which may still be running in the next decision cycle but are requested to stop. For that reason, the state settings are included in this prototype and are not to be confused with the FSM pattern.

5.1.1 State Settings

As hinted in Section 3.1 using states for visual debugging and including them in conditional operations for decision-making is an advancement for a generic AIS and necessary to solve the issue finding already running actions in the separate scheduling systems.

The state settings derive from an abstract class *BaseLemma*, which includes fields such as the label (string) for description, a *labelHash* (ulong) generated from the string for faster comparisons and an optional parent of type *BaseLemma*. The parent field enables to group state settings in a hierarchy for potentially less maintenance for the AI designer and less logical checks if entire state groups can be found within one check.

5.1.2 Condition Settings

Condition settings offer a mechanism to validate or invalidate a certain given input in a modular and expandable manner by using them as an optional attachment for other settings, reducing the number of nodes and layers in a tree unlike the conventional behavior tree pattern does (see Section 3.3).

The expandability is guaranteed with an inheritable virtual method as displayed in Listing 5-1. Said method takes two parameters of type *OneAIEntity* (class highlighted in Section 5.6), one being the target for the Condition to run on and the other being the Condition requester. Both parameters may be the same object in reference (target = requester) which is generally the case for the decision-making process discussed in Section 5.5 but can differ in specific settings and actions, such as in the implementations of spotting mechanics (target = spotted, requester = observer). The requester *OneAIEntity* parameter is most of the time required for debugging purposes (visualize current picked decision), as it is the case for the graph editor discussed in Section 5.7. The *OneAIConditionResult* object is a wrapper class returning the result of the Condition method with a bool value, indicating the overall success state of the Condition and additionally providing a score value of type float for advanced decision-making options (see Section 5.1.4).

```

18  public virtual OneAIConditionResult IsConditionMetFor(OneAIEntity targetOneAIEntity,
19      OneAIEntity conditionsRequesterEntity)
20  {
21      return new OneAIConditionResult(success:targetOneAIEntity != null);
22  }
    
```

Listing 5-1: OneAI Base Condition Method

OneAIStateCondition, a specific implementation of such a Condition class is displayed in Listing 5-2 with the purpose to check the required or not allowed (depending on the Condition configuration) state settings against the current applied states on the target *OneAIEntity*.

```

45  public override OneAIConditionResult IsConditionMetFor(OneAIEntity targetOneAIEntity,
46      OneAIEntity conditionsRequesterEntity)
47  {
48      bool inState = this.stateListWrapper.list.Count < 1
49          || BaseLemma.LemmasInLemmas(victimLemmas:targetOneAIEntity.currentStates.ToArray(),
50              blockerLemmas:this.stateListWrapper.list.ToArray(), this.checkStateHierarchy);
51
52      inState = this.invert ? !inState : inState;
53
54      this.MarkInGraph(conditionsRequesterEntity, inState);
55
56      return new OneAIConditionResult(inState, inState ? 1 : 0);
57  }
    
```

Listing 5-2: OneAI State Condition

The *OneAIBaseCondition* class can furthermore be used to group other Conditions together and handle the summarized results specifically, for instance:

- *ORConditionsContainer* (Any condition must pass)
- *ANDConditionsContainer* (All conditions must pass)

```

8 public abstract class OneAIBaseConditionForDescriptors<DescriptorType>
9     : OneAIBaseCondition where DescriptorType : OneAIBaseDescriptor
10    {

```

Listing 5-3: OneAI Base Condition For Descriptors

To inject specific runtime data required from an *OneAIEntity* and make use of the local blackboard pattern (see Section 3.7), an abstract *OneAIBaseConditionForDescriptors* wrapper class is written and illustrated in Listing 5-3. A Descriptor type generic for its inherited specialized Condition classes is required with this implementation and is furthermore necessary for the mapping mechanism examined in Section 5.2 in depth.

The *OneAIBaseDescriptor* class is a runtime instance-based data container, editable in the Unity Inspector and exists for the purpose to provide a modular architecture for data-injection, specifically for Conditions and is highlighted in Section 5.4.

Inheriting from this wrapper class and making use of the mapping mechanics (line 33-34), for example reading and interpreting the health of an *OneAIEntity* is illustrated in Listing 5-4.

```

30 public override OneAIConditionResult IsConditionMetFor(OneAIEntity targetOneAIEntity,
31     OneAIEntity conditionsRequesterEntity)
32    {
33        List<OneAIHealthDescriptor> healthDescriptors =
34            targetOneAIEntity.GetDescriptorsByHash<OneAIHealthDescriptor>(this.descriptorTypeSupportedHash);
35
36        //no health descriptor = immortal!
37        if (healthDescriptors.Count < 1)
38        {
39            this.MarkInGraph(conditionsRequesterEntity, this.validOnNoMatchingDescriptorFound);
40            return new OneAIConditionResult(this.validOnNoMatchingDescriptorFound);
41        }
42
43        OneAIHealthDescriptor oneAIHealthDescriptor = healthDescriptors[0];
44
45        bool isValid = oneAIHealthDescriptor.Health <= this.healthThreshold;
46        isValid = thresholdIsMaxAllowedInclusive ? isValid : !isValid;
47
48        this.MarkInGraph(conditionsRequesterEntity, isValid);
49        return new OneAIConditionResult(isValid);
50    }

```

Listing 5-4: OneAI Health Descriptor Condition

5.1.3 Action Settings

Action settings represent the leaf settings of the AI tree, containing information about the execution frequency, life span, and input data for the resulting action, managed by the scheduler.

Since each action must be maintained inside a scheduler, the same base action setting applies for every custom (specialized) action.

The base action setting, as illustrated in Figure 5-1 provides relevant information about the resulting action for the scheduler and the decision-making system, such as:

- Infinite or limited repetition
- Delay of execution, as well as the exact timings
- Max allowed actions in one scheduler at once sharing this exact setting (reference)
- The priority, indicating the execution order if multiple actions are present within the same scheduler
- The Condition to either run or ignore the action
- The Condition to either include or exclude this action from the decision result
- The disposal requirements indicating when and under which Conditions the action should terminate
- The state setting represented by this action (setting)
- States and actions, which should be removed upon entering the scheduler
- States and actions, which either prevent this action from entering the scheduler or terminate this action when already running

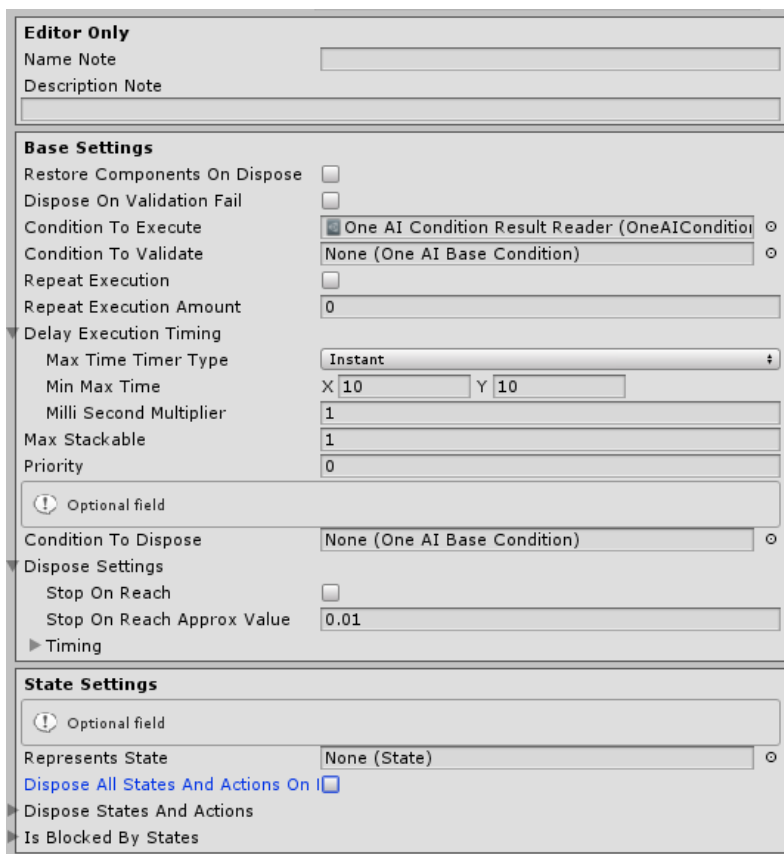


Figure 5-1: OneAI Base Action Settings

Terminating already running (forever repeating) actions, which were chosen in a previous decision result but are invalid in the present decision result with the help of state settings, solves the issue not knowing when and which action to stop, mentioned in the beginning of Chapter 5. Furthermore, this approach offers an EDA by triggering the termination of an action, which is in this case faster than checking (each frame) if the action should be terminated.

5.1.4 Decider Settings

For this AIS prototype a mixed tree pattern is chosen, which includes implementations of both the behavior and decision tree (see Section 3.2 and 3.3).

To keep the tree structure as simple and clean as possible, the branch nodes only contain children either of type leaf nodes (actions) or other branch nodes but not both at the same time, which is an implementation used primarily in the decision tree pattern. The decorator task used in the behavior tree pattern is removed completely from this mixed tree pattern since its functionality is replaced by the separate scheduling system combined with the action settings.

Figure 5-2 and Figure 5-3 highlight the Decider setting in the Unity Inspector, one in form of a leaf Decider (branch-leaf), only containing action settings of the base type *OneAIBaseSettings* as children and the other a branch Decider, referencing other Decider branches as its children.

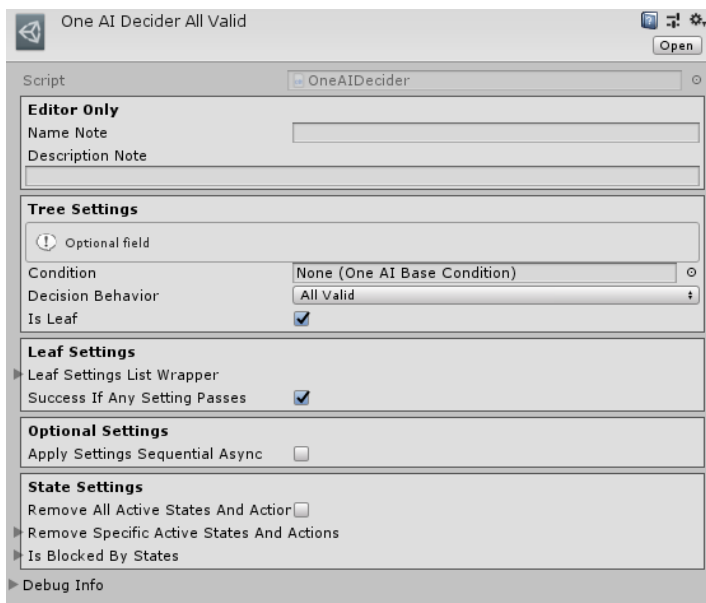


Figure 5-2: OneAI Branch-Leaf Decider

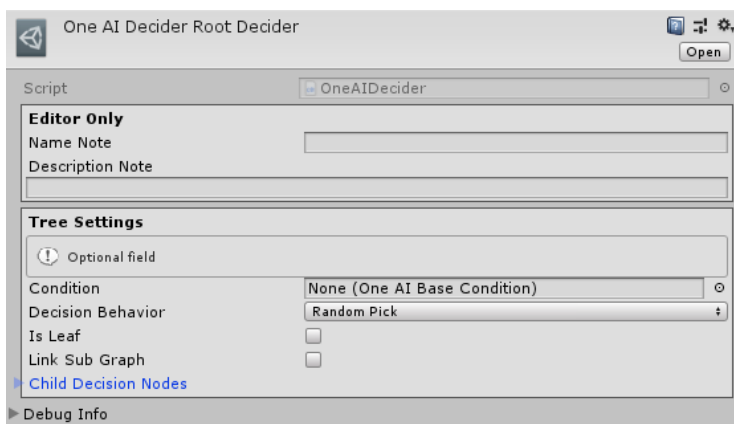


Figure 5-3: OneAI Branch Decider

The Decider setting setup as a branch-leaf offers an option to queue the action settings list in a sequential manner, meaning that the action settings are not immediately added to their respective schedulers at once but rather get added after the previous action terminates. This is done in the independent class *AsyncSettingsQueueHandler* which subscribes to the current active action’s disposal event.

If the Decider setting is setup as a branch node, it can either reference further Decider settings as direct referenced children, or an entire (sub) graph, referencing another root Decider node from a (different) graph. This option increases the maintainability as stated in the requirements Section 2.3 and provides a modular approach for the AI designer by subdividing a complex AI behavior into reusable sub behaviors.

Listing 5-5 displays the method of the Decider setting, responsible for filtering valid branch-leaf Decider nodes (containing only action settings as children). This method can be inherited to include custom decision-making algorithms similar to the composite task structure from the behavior tree pattern. The returning list (line 258) with items of type *ApplyDecisionNodeResult* store each traversed Decider node, including if the traversal was successful in the first place, as well as containing information about validated and invalidated action settings, all of which for debugging reasons. The *OneAIEntity* parameter (line 259) is used as a target and a requesting entity for (optional) Conditions. The ref list (line 260) with items of type *OneAIDecider* is the decision result, containing only branch-leaf Deciders with action settings as children to be handed over to the respective scheduling systems via the decision system highlighted in Section 5.5.

```

258     public virtual List<ApplyDecisionNodeResult> FindValidLeafNodes(
259         OneAIEntity oneAIEntity,
260         ref List<OneAIDecider> leafNodes)
    
```

Listing 5-5: OneAI Tree Traversal Method

Listing 5-6 shows an Enum used in this Decider setting to choose from a set of possible decision-making and tree traversal techniques and is integrated in the base *OneAIDecider* method, as showcased in Listing 5-5.

```

95     public enum DecisionBehavior
96     {
97         FirstValid = 0,
98         AllValid = 1,
99         RandomPick = 2,
100        HighestScore = 3
101    }
    
```

Listing 5-6: OneAI Decisions Enum

Although it is possible to integrate custom decision-making algorithms, the current implementation requires the AI programmer to also handle the tree traversal and optional condition calls when overriding this method. Refactoring the tree traversal and the Condition checks into separate also overridable methods would solve this issue and is due to the limited amount of time considered to be integrated in the next development iteration of this AIS prototype.

5.2 Mapping Types

In order to provide an inheritance-based architecture for the core AIS components, as well as specific data injection for actions and schedulers, an automated mechanism that maps custom created and therefore yet unknown and of each other dependent classes, while avoiding performance heavy reflection calls at runtime is a necessity for this generic AIS prototype.

Figure 5-4 illustrates the class relations as follows:

- Custom Setting A and Custom Setting B inherit from Base Setting
- Custom Engine A and Custom Engine B inherit from Base Engine
- Custom Action A and Custom Action B inherit from Base Action
- Therefore, the Base Setting, Base Action and Base Engine classes do not share the same parent
- Custom Setting A and Custom Engine A both map to Custom Action A
- Custom Setting B and Custom Engine B both map to Custom Action B

The mapping issue, using the illustration of Figure 5-4 is described in the following example:

- Custom Setting A and B are dynamically added in the same list LS with items of type Base Setting
- Custom Engine A and B are stored in the same list LE with items of type Base Engine
- The goal is to map a Custom Setting with a Custom Engine, both of which dependent of the same exact Custom Action
- Following mappings are allowed:
 - Custom Setting A and Custom Engine A
 - Custom Setting B and Custom Engine B
- Iterating through each element of LS and LE, how to find the allowed mappings?

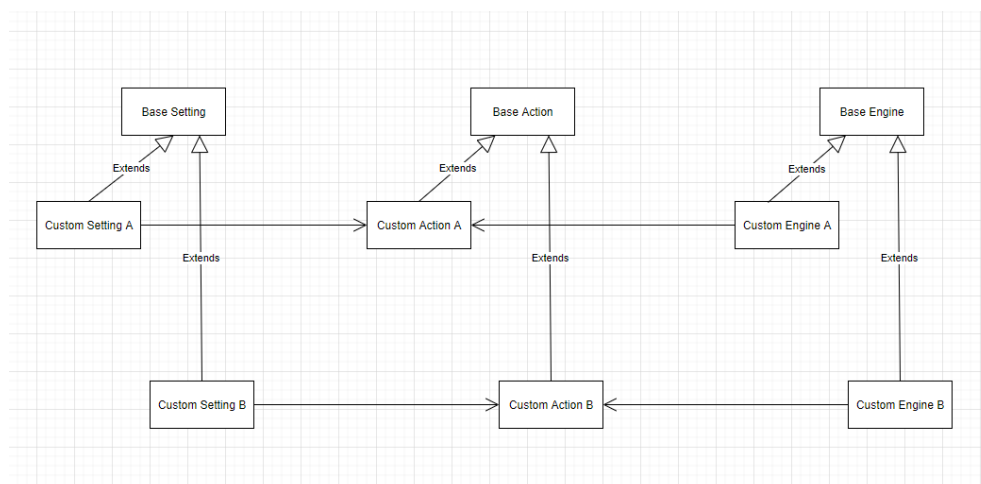


Figure 5-4: Generic Mapping Issue

The inherited class knows its dependent type via its pre-defined generic, which is restricted to inherit from the dependent type only, as it is showcased in line 17 of Listing 5-7 and line 13 of Listing 5-8. With that, the generic type can be stored as a variable, offering the ability to compare the dependent generic types via reflection, solving the mapping issue in the first step. This procedure though comes with an expensive performance cost and is not applicable if mapping multiple dynamic instances is required often at runtime [32].

```

14 public abstract class OneAIEntityBaseEngine<ActionType, BlackboardType>
15     : OneAIBaseNotesMono, IBaseEngineWrapper
16     where BlackboardType : IBlackboard
17     where ActionType : BaseAction<BlackboardType>
18     {

```

Listing 5-7: OneAI Base Engine

```

10 public abstract class OneAIBaseSettingsForEngine<ActionType, BlackboardType>
11     : OneAIBaseSettings
12     where BlackboardType : IBlackboard
13     where ActionType : BaseAction<BlackboardType>

```

Listing 5-8: OneAI Base Settings For Engine

To avoid this performance issue, the caching procedure as stated below is established:

- Get the base type of the desired other class via reflection at compile time or at worst at (runtime) initialization
- Extract the name of the desired class and store the string value
- Convert the string into a hash integer value and store it

After the caching is done, the generic type comparison via reflection can be avoided with the much faster hash comparison. One risk, that comes with hash generation is, if the hash generation algorithm is handled at the OS level and returns different outputs depending on the current used OS, which is a problem when multiple developers work with the same Unity project, but not on the same OS. This issue can be avoided by implementing reliable open-source hash generation algorithms or automatically recalculating the hashes upon starting the Unity project or built game.

Another risk of converting string values to hash integers is, that hash collision of different string contents is a possibility, leading to false confirmations. The probability for a hash collision, specifically for this use case (names of classes) is not high enough (up to 4,294,967,295 possible hash values for a 32bit Integer) and the performance boost is too beneficial to discard the hash comparison technique.

Listing 5-9 displays the implementation of a reflection call in line 264 to 265 and hash generation in line 270 for every scheduler (OneAI-Engine) type, inheriting the OneAI-BaseEngine to resolve the hash of its supported generic action type. The *ReflectionHelper* method used in this illustration in line 264 returns the first found generic type, after traversing up to the OneAI-BaseEngine type, for the reason to ensure the correct generic argument. This is because custom written child OneAI-Engine classes might include additional generics or not use any generics at all in their class header, causing an invalid hash in the final result (line 264 and 270).

```

260     protected virtual void UpdateSupportedActionType()
261     {
262         if (string.IsNullOrEmpty(this.ActionTypeSupported))
263         {
264             this.ActionTypeSupported = ReflectionHelper.GetFoundGenericTypeOfParentClassAtIndex(descendant:this.GetType(),
265             parentToFind:typeof(OneAIBaseSettingsForEngine<BaseAction<IBlackboard>, IBlackboard>)).Name; // string
266         }
267         //safety update for different hash results on different machines
268         else
269         {
270             this.actionTypeSupportedHash = actionTypeSupported?.GetHashCode() ?? -1;
271         }
272     }
273

```

Listing 5-9: OneAI Automatic Type To Hash Generation

Using this mapping mechanic enables any AI developer to build custom AI behaviors and settings, expanding from and still interacting with the core system of this prototype. Furthermore, **securing the scalability and performance requirements** from Section 2.1 and 2.2.

5.3 Blackboards, Actions and Engines

The previous Section 5.2 explains the relations between the action settings, actions and OneAI-Engines, as well as solving the mapping issue of inherited types. Based on this, the detailed implementation of the OneAI-Engine in combination with the action and the blackboard pattern is examined in this section, highlighting the separated action execution from the decision-making algorithms.

Listing 5-10 represents the wrapper interface, which is implemented by the *BaseAction* class, illustrated in Listing 5-11. The wrapper interface is necessary to store any type of action in a variable (or list), without pre-defining the *BlackboardType* generic, thus avoiding syntactical and compilation errors in external classes.

The relevant properties for the OneAI-Engine are:

- *IsDone*, indicating if the action should be discarded
- *RunCount*, indicating how often this action is run and in combination with the settings, how often the action is allowed to run
- *InvalidRunsAmount*, indicating how often the validation failed of this action and in combination with the settings, how often the action is allowed to invalidate
- *Settings* and *StoppingSettings* are used to handle the actions termination and keepalive requirements
- *RunDelayTiming*, indicating the execution frequency of this action


```

8  public interface IBaseActionWrapper : IDisposable
9  {
10     event Action<IBaseActionWrapper> OnActionDisposed, OnActionInitialized;
11
12     /// <summary>
13     /// Is the action done yet?
14     /// </summary>
15     bool IsDone { get; set; }
16
17     bool Success { get; }
18
19     int RunCount { get; set; }
20
21     int InvalidRunsAmount { get; set; }
22
23     IBaseEngineWrapper Engine { get; }
24
25     OneAIBaseSettings Settings { get; }
26
27     Timing RunDelayTiming { get; }
28
29     Stopping StoppingSettings { get; }
30 }
31

```

Listing 5-10: OneAI Action Wrapper Interface

```

10 public abstract class BaseAction<BlackboardType>
11     : IBaseActionWrapper where BlackboardType : IBlackboard

```

Listing 5-11: OneAI Base Action

The *Success* bool property is mainly used in the *AsyncSettingsQueueHandler* class (mentioned in Section 5.1.4) to indicate if the action queue should abort or continue on failure or success.

The *Engine* property is a helper property to offer a direct reference to the host OneAI-Engine, containing the action instance.

Listing 5-12 is an excerpt of the *BaseAction* class showcasing abstract methods necessary for any action type to be handled by the OneAI-Engine, as shown in the illustration of Listing 5-13 in lines 592 and 598. This scheduling algorithm of the OneAI-Engine is managing an action’s execution frequency (line 585), termination (line 603 and 613) and validation (line 592).

```

58  Frequently called  1 usage  11 overrides  Nigg
public abstract void Initialize(BlackboardType blackboard);
59
60  Frequently called  1 usage  20 overrides  Nigg
public abstract bool Validate(BlackboardType blackboard);
61
62  /// <summary>
63  /// The actions work
64  /// </summary>
65  Frequently called  1 usage  30 overrides  Nigg
public abstract void Run(BlackboardType blackboard);
66
67  /// <summary>
68  /// Restores the previous state
69  /// </summary>
70  Frequently called  1 usage  25 overrides  Nigg
public abstract void Restore(BlackboardType blackboard);
71
72  Frequently called  8 usages  4 overrides  Nigg
public virtual void Dispose(){...}

```

Listing 5-12: OneAI Base Action Abstract Methods

```

580  private void RunActions(BlackboardType blackboard)
581  {
582      foreach (ActionType baseAction in _actions)
583      {
584          //check if the delay is kicking in
585          if (!baseAction.RunDelayTiming.CheckTimeReachedOrExceeded())
586              continue;
587
588          //reset delay
589          baseAction.RunDelayTiming.CheckAgainstTime = DateTime.Now;
590
591          //does the action validate for execution internally?
592          if (baseAction.Validate(blackboard) &&
593              //is the optional condition met to validate the action externally?
594              (baseAction.Settings.conditionToValidate == null
595              || baseAction.Settings.conditionToValidate.IsConditionMetFor(this.OneAIEntity, this.OneAIEntity).success))
596          {
597              //run the action
598              baseAction.Run(blackboard);
599          }
600          else if (baseAction.Settings.disposeOnValidationFail)
601          {
602              if (++baseAction.InvalidRunsAmount > baseAction.Settings.maxAllowedFailedValidations)
603                  this.RemoveActionsQueue(CollectionHelper.ListOfOne(baseAction));
604              continue;
605          }
606
607          //check for repetition limits and stopping settings
608          ActionHelper.UpdateIsDoneGeneral(baseAction, this.OneAIEntity);
609
610          //the action has finished its work, so we can discard it
611          if (baseAction.IsDone)
612          {
613              this.RemoveActionsQueue(CollectionHelper.ListOfOne(baseAction));
614          }
615      }
616  }

```

Listing 5-13: OneAI Base Engine Run Actions

Listing 5-14 displays the *IBaseActionWrapper* interface implemented by the *OneAIEntityBaseEngine* class in Listing 5-15. The wrapper interface exists to store any OneAI-Engine type without a specified generic type in a variable (see Section 5.6), which is the same reason the *IBaseActionWrapper* interface is created (see Listing 5-10 and Listing 5-11).

```

6  | public interface IBaseEngineWrapper
7  | {
8  |     string ActionTypeSupported { get; }
9  |
10 |     int ActionTypeSupportedHash { get; }
11 |
12 |     OneAIEntity OneAIEntity { get; }
13 |
14 |     ApplyActionResult ApplySettings(OneAIBaseSettings settings);
15 |
16 |     void RemoveAllActionsAndStatesQueue();
17 | }
18 |

```

Listing 5-14: OneAI Base Engine Wrapper Interface

```

14 | public abstract class OneAIEntityBaseEngine<ActionType, BlackboardType>
15 |     : OneAIBaseNotesMono, IBaseEngineWrapper
16 |     where BlackboardType : IBlackboard
17 |     where ActionType : BaseAction<BlackboardType>
18 | {

```

Listing 5-15: OneAI Base Engine Header

Listing 5-16 illustrates the generic action types in queues, to either enter (line 59) or leave (line 60) the current active executing list (line 57) and is handled in the *Tick* method provided in the *OneAIEntityBaseEngine* (see Listing 5-17):

- The *_actionsAddQueue* is handled in line 395 within the *Tick* method
- The *_actionsRemoveQueue* is handled in line 392 within the *Tick* method
- Executing the actions as illustrated in Listing 5-13 is handled in line 400 within the *Tick* method

```

57     protected List<ActionType> _actions = new List<ActionType>();
58
59     protected Queue<ActionType> _actionsAddQueue = new Queue<ActionType>();
60     protected Queue<ActionType> _actionsRemoveQueue = new Queue<ActionType>();

```

Listing 5-16: OneAI Base Engine Action Queues

```

383     /// <summary>
384     /// The Tick that may be called in custom or other pre defined update methods
385     /// </summary>
386     public virtual void Tick(BlackboardType blackboard)
387     {
388         //modify actions before adding new ones (to not mess up sorting)
389         this.ModifyActions(blackboard);
390
391         //remove actions
392         this.RemoveActions(blackboard);
393
394         //add actions that are in queue
395         if (this.AddActions(blackboard))
396             //sort the actions list
397             this.SortActionsByPriorities(blackboard);
398
399         //Run the actions
400         this.RunActions(blackboard);
401     }

```

Listing 5-17: OneAI Base Engine Tick Method

Listing 5-19 showcases an example of a custom OneAI-Engine hosting actions of type *BaseFollowAction* and provides the specified *FollowBlackboard* type displayed in Listing 5-18, implementing the local blackboard pattern explained in Section 3.7 in a specific use case, avoiding direct casts and type conversions.

```

6   public class FollowBlackboard : IBlackboard
7   {
8       public List<OneAIEntity> targets = new List<OneAIEntity>();
9
10      public OneAIEntityDestinationContainer pickedTargetsDestinationContainer;
11
12      public float deltaTime = 1;
13  }

```

Listing 5-18: OneAI Follow Blackboard Example

```

16  public abstract class OneAIEntityBaseFollowEngine
17      : OneAIEntityBaseEngine<BaseFollowAction, FollowBlackboard>

```

Listing 5-19: OneAI Follow Engine Example

Listing 5-20 highlights the usage of the *Tick* method defined in Listing 5-17 in combination with custom blackboard data injection, handled in the *BaseFollowAction* instances. Listing 5-21 is an example of a specific OneAI-Engine using Unity’s Update system to call the *Tick* method in line 12. This is evidence for an expandable scheduling system not restricting the user to a pre-defined and pre-compiled update frequency.

```

108  public override void Tick(FollowBlackboard blackboard)
109  {
110      blackboard.pickedTargetsDestinationContainer = this.resultEntitiesDestinationContainer;
111      base.Tick(blackboard);
112  }

```

Listing 5-20: OneAI Follow Engine Tick Usage

```

6   public class OneAIEntityFollowUpdateEngine : OneAIEntityBaseFollowEngine
7   {
8       // Update is called once per frame
9       protected void Update()
10      {
11          this.Blackboard.deltaTime = Time.deltaTime;
12          this.Tick(this.Blackboard);
13      }
14  }

```

Listing 5-21: OneAI Tick In Update Method

Listing 5-22 showcases the overridden *ApplySettings* method, where the specific follow actions are instantiated (visible from line 127 to 146) based on the given settings parameter. The base *ApplySettings* called in line 116 at first validates the settings by checking if any preventing states are present within the *OneAIEntity*. The resulting action is added into the local queue variable. Each new (custom) OneAI-Engine must implement the action instantiation in a similar manner to convert custom settings into the respective custom actions. Since rewriting the same boilerplate code leads to bad maintainability for the AI programmer, a code generation tool is added to this prototype (outlined in Section 5.8).

```

114 public override ApplyActionSettingResult ApplySettings(OneAIBaseSettings settings)
115 {
116     ApplyActionSettingResult applySettingResult = base.ApplySettings(settings);
117
118     if (!applySettingResult.success)
119     {
120         return applySettingResult;
121     }
122
123     OneAIFollowSettings followSettings = (OneAIFollowSettings)settings;
124     //the next action to be added into the agentai engine
125     BaseFollowAction followAction = null;
126
127     switch (followSettings.followType)
128     {
129         case OneAIFollowSettings.FollowType.TailTarget:
130             followAction = new FollowTailAction(followSettings, followEngine:this);
131             break;
132         case OneAIFollowSettings.FollowType.PredictTarget:
133             break;
134         case OneAIFollowSettings.FollowType.RandomPoint:
135             followAction = new FollowRandomAction(followSettings, followEngine:this);
136             break;
137         case OneAIFollowSettings.FollowType.JustLookAtTarget:
138             followAction = new JustLookAtFollowAction(followSettings, followEngine:this);
139             break;
140         case OneAIFollowSettings.FollowType.SimpleFlankTarget:
141             followAction = new SimpleFlankTargetAction(followSettings, followEngine:this);
142             break;
143         case OneAIFollowSettings.FollowType.Boid:
144             followAction = new BoidFollowAction(followSettings, followEngine:this);
145             break;
146     }
147
148     this.AddActionsQueue(CollectionHelper.ListOfOne(followAction));

```

Listing 5-22: OneAI Follow Engine Action Creation

Listing 5-23 displays the in the OneAI-Engine implemented EDA, terminating running actions independent of the decision-making system, every time a state is either added or removed. This method solves the issue mentioned in Chapter 5, not knowing how and when to address obsolete but still executing actions by the schedulers in a performant fashion.

```

142     protected virtual void OnAIEntityStateChanged(IBaseEngineWrapper engineWrapper, State state, bool added)
143     {
144         //YES I AM AWARE
145         #pragma warning disable
146         //dont get notified by self
147         if (engineWrapper != this && state != null)
148         {
149             if (added)
150             {
151                 //check if any action shall be blocked by the state
152                 this.ActionsInQueueAreBlockedByState(state);
153                 this.ActionsActiveBlockedByState(state, this.GetCopyOfActiveActionsList());
154             }
155             else
156             {
157                 //Debug.Log(state.label);
158
159                 this.RemoveActionsQueue(
160                     this.GetCopyOfActiveActionsList().FindAll(act:ActionType =>
161                         act.Settings.representsState != null && act.Settings.representsState.InLemma(state)));
162             }
163         }
164     }

```

Listing 5-23: OneAI Base Engine EDA

5.4 Descriptors

As already hinted at in Section 5.1.2, the Descriptor class is mainly used as a specific data container for Conditions but can also be used in combination with OneAI-Engines, actions or any other architecture.

Unlike the blackboard implementation combined with actions and OneAI-Engines, the Descriptor class is an optional Unity Component and is not tied to any OneAI-Engine or action. Since the *OneAIEntity* only contains state settings to describe itself but not any other form of instance-based data containers, the Descriptor class is created as a dynamic extension to the *OneAIEntity*, offering a modular approach to include custom runtime data per instance.

Listing 5-24 is an example implementation of the *OneAIBaseDescriptor*, storing health as a local float variable for the representing *OneAIEntity* and is referenced in the health Descriptor Condition, as already shown in Listing 5-4 of Section 5.1.2.

```

14     public class OneAIHealthDescriptor : OneAIBaseDescriptor
15     {
16         # Nigg
17         public event Action<float, float> OnHealthChanged;
18         public OnHealthChangedUnityEvt OnHealthChangedUnityEvt = new OnHealthChangedUnityEvt();
19
20         [SerializeField]
21         protected float health = 100; // "200"
22
23         # 1 usage # Nigg
24         public float Health{...}
25     }

```

Listing 5-24: OneAI Health Descriptor Example

5.5 Decision System

In order to trigger the decision algorithms established in the Decider settings (see Section 5.1.4) on an agent instance at runtime, the Decision System is built as a Unity Component for the respective *OneAIEntity*.

The decision-making algorithms of the Decider settings are handled in line 141 of Listing 5-25. The ref list *leafNodes* from line 139 stores the gathered branch-leaf Decider nodes, which are iterated through in line 148, to be potentially included in or excluded from the final decision. Line 191 handles the sequential setup Decider nodes with their child action settings, by injecting the actions into the designated *AsyncSettingsQueueHandler* objects, as mentioned in Section 5.1.4.

```

131     protected virtual void Tick(OneAIDecider rootAIDecider)
132     {
133         if (this.makeDecision)
134         {
135             this.makeDecision = false;
136
137             this.OnPrepareNextDecision?.Invoke();
138
139             List<OneAIDecider> leafNodes = new List<OneAIDecider>();
140             List<ApplyDecisionNodeResult> decisionNodeResults
141                 = rootAIDecider.FindValidLeafNodes(this.oneAIEntityToDecide, ref leafNodes);
142
143             List<ApplyDeciderResult> applyStorageResults = new List<ApplyDeciderResult>();
144
145             List<AsyncSettingsQueueHandler> newAsyncSettingsStorageHandlers =
146                 new List<AsyncSettingsQueueHandler>();
147
148             foreach (OneAIDecider leaf in leafNodes){...}
189
190             //add all previous handlers, that were not in the current decision but desire continuation regardless
191             this.asyncSettingsStorageHandlers.ForEach(handler =>
192                 {...});
205
206             //apply the new async handlers list
207             this.asyncSettingsStorageHandlers = newAsyncSettingsStorageHandlers;
208
209             //leave old reference in case the entity was cloned/ duplicated!
210             this.currentDecision = new Decision();
211
212             this.CurrentDecision.leafStorageNodes = leafNodes;
213             this.CurrentDecision.decisionNodeResults = decisionNodeResults;
214             this.CurrentDecision.applyStorageResults = applyStorageResults;
215
216             this.OnDecisionMade?.Invoke(this.CurrentDecision);
217         }
    
```

Listing 5-25: OneAI Decision System Tick Method

The foreach loop in Listing 5-26 contains a switch statement, handling the filtering of the child action settings from the current iterated leaf Decider node with the Enum illustrated in Listing 5-6. Using a switch statement to filter the action settings is limiting the expandability for custom filtering algorithms, unless the entire *Tick* method is overridden. This issue is similar to the expandability issue described in the Decider settings (see Section 5.1.4) and is addressed in the next development iteration of this AIS prototype, due to the limited development time.

```

148     foreach (OneAIDecider leaf in leafNodes)
149     {
150         ApplyDeciderResult applyDeciderResult;
151
152         switch (leaf.decisionBehavior)
153         {
154             case OneAIDecider.DecisionBehavior.AllValid:
155                 applyStorageResults.Add(item:this.HandleAllSettingsGroup(leaf, ref newAsyncSettingsStorageHandlers));
156                 break;
157             case OneAIDecider.DecisionBehavior.FirstValid:
158                 for (int i = 0; i < leaf.leafSettingsListWrapper.list.Count; i++)
159                 {
160                     applyStorageResults.Add(
161                         applyDeciderResult = this.HandlePickedSetting(leaf.leafSettingsListWrapper.list[i],
162                             leaf));
163
164                     //we only want to apply the first valid @group (selector tree behaviour)
165                     if (applyDeciderResult.success)
166                         break;
167                 }
168
169                 break;
170             case OneAIDecider.DecisionBehavior.HighestScore:
171                 List<OneAIBaseSettings> settingsList =
172                     new List<OneAIBaseSettings>(leaf.leafSettingsListWrapper.list);
173                 settingsList.Sort(comparer:new OneAISettingsScoreSorter(oneAIEntityToDecide, oneAIEntityToDecide));
174
175                 applyStorageResults.Add(item:this.HandlePickedSetting(
176                     settingsList.FirstOrDefault(),
177                     leaf));
178                 break;
179             case OneAIDecider.DecisionBehavior.RandomPick:
180
181                 int randomPickIndex = UnityEngine.Random.Range(0, leaf.leafSettingsListWrapper.list.Count);
182                 //Debug.Log(randomPickIndex);
183                 applyStorageResults.Add(item:this.HandlePickedSetting(
184                     leaf.leafSettingsListWrapper.list[randomPickIndex],
185                     leaf));
186                 break;
187         }
188     }
    
```

Listing 5-26: OneAI Decision System Filter Deciders

Listing 5-27 represents the method for single picked action settings called from the foreach loop and in line 285 passes the setting to the host *OneAIEntity* instance, where the setting is mapped to the correct OneAI-Engine using the mechanism described in Section 5.2 and is implemented in Section 5.6.

```

282 public virtual ApplyDeciderResult HandlePickedSetting(OneAIBaseSettings setting, OneAIDecider leaf)
283 {
284     ApplyDeciderResult applyDeciderResult =
285         this.oneAIEntityToDecide.ApplyLeafDecider(leaf, CollectionHelper.ListOfOne(setting));
286
287     leaf.MarkInGraph(this.oneAIEntityToDecide, applyDeciderResult.success);
288
289     return applyDeciderResult;
290 }

```

Listing 5-27: OneAI Decision System Transfer Setting

Listing 5-25 demonstrates the core logic of the Decision System located within the *Tick* method, which can be called at custom frequencies, events, or in the default Unity Update method as it is done in Listing 5-28, similar to the OneAI-Engine system pictured in Listing 5-21 of Section 5.3.

```

4 public class DecisionSystemUpdate : DecisionSystem
5 {
6     Event function Nigg
7     protected virtual void Update()
8     {
9         this.Tick(this.nodeGraph.rootDecider);
10    }

```

Listing 5-28: OneAI Decision System Update Example

5.6 OneAIEntity

The *OneAIEntity* object (see Listing 5-29) is the centralized AI Component containing references to Descriptors and *IBaseEngineWrapper* interfaces, therefore not needing to specify any generic for different types of specific OneAI-Engines. Furthermore, this class offers a list with items of type state setting to visualize which actions might be running and in which state the current agent GameObject is in.

```

23 public class OneAIEntity : OneAIBaseNotesMono
24 {
25     public event Action<IBaseEngineWrapper, State, bool> OnStateChanged;
26     public OnStateChangedUnityEvent OnStateChangedUnityEvent; 8 methods
27
28     [ReadOnly]
29     public List<OneAIBaseDescriptor> descriptors = new List<OneAIBaseDescriptor>();
30     public List<IBaseEngineWrapper> oneAIEngines = new List<IBaseEngineWrapper>();
31
32     /// <summary>
33     /// The current leaf states, this entity represents
34     /// </summary>
35     [Expandable]
36     public List<State> currentStates = new List<State>(); Serializable
    
```

Listing 5-29: OneAI Entity Fields

Listing 5-31, a cutout from the *ApplyLeafDecider* method called from the Decision System (see line 285 in Listing 5-27), showcasing how the OneAI-Engine is mapped to the action setting using the mapping mechanic from Section 5.2, starting in line 180 and is fully revealed in Listing 5-30.

```

92 public virtual IBaseEngineWrapper GetEngineByHash(int hash)
93 {
94     //find the correct engine
95     IBaseEngineWrapper foundEngine = this.oneAIEngines.Find(match: (engine) =>
96         engine.ActionTypeSupportedHash == hash
97     );
98
99     return foundEngine;
100 }
    
```

Listing 5-30: OneAI Entity Hash To Engine Method

```

179 //engine specific apply settings (find correct engine)
180 IBaseEngineWrapper foundEngine = this.GetEngineByHash(settings.ActionTypeSupportedHash);
181
182 //keep track if a setting fails
183 bool allSettingsPassed = foundEngine != null;
184 if (allSettingsPassed)
185 {
186     ApplyActionSettingResult applySettingResult = foundEngine.ApplySettings(settings);
187 }
    
```

Listing 5-31: OneAI Entity Get Engine By Hash

Listing 5-32 exposes a similar technique used in Listing 5-30, with the difference that a direct cast is necessary to return the list with a specified Descriptor item type, to read or write from a specific Descriptor class (see Listing 5-4 in Section 5.1.2), unlike simply calling an abstract method as it is the case with the *IBaseEngineWrapper* interface.

This setup offers a performant, scalable and modular approach for adding custom and optional data containers (Descriptors) and for mapping custom action settings to their respective custom OneAI-Engines.

```

70 public virtual List<T> GetDescriptorsByHash<T>(int hash) where T : OneAIBaseDescriptor
71 {
72     //Unlike the engines, the descriptor is required as the exact type to retrieve its information.
73
74     List<T> foundDescriptors = new List<T>();
75     for (int i = 0; i < this.descriptors.Count; i++)
76     {
77         OneAIBaseDescriptor desc = this.descriptors[i];
78         if (desc == null)
79         {
80             this.descriptors.RemoveAt(i--);
81             continue;
82         }
83         //Debug.Log("desc: " + desc.DescriptorTypeSupportedHash);
84
85         if (desc.DescriptorTypeSupportedHash == hash)
86             foundDescriptors.Add((T) desc);
87     }
88
89     return foundDescriptors;
90 }

```

Listing 5-32: OneAI Entity Hash To Descriptor Method

Figure 5-5 features the *OneAIEntity* rendered in the Unity Inspector, setup in a way to call the Decision System, whenever its state changes (see the *OnStateChangedUnityEvent* at the top), providing an example use case of an EDA.

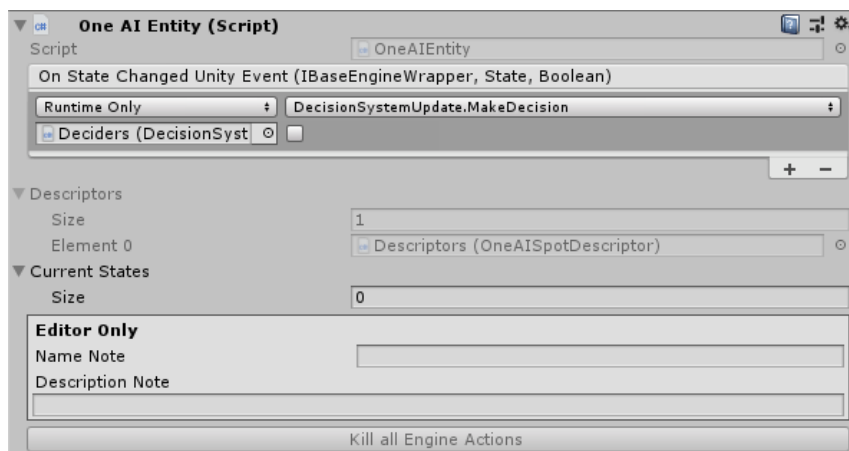


Figure 5-5: OneAI Decision System Component

5.7 Graph Editor

As identified in the test cases of the evaluated asset packs in Chapter 4, a graph editor is a powerful tool to illustrate the behavior of an AI in an organized and compact manner, while offering abilities to add, remove and modify the AI behavior quickly and efficiently. A well build graph editor increases the usability, user experience and maintainability drastically.

Based on the observations from Chapter 4, following graph editing features are essential for this AIS prototype:

- Render a background grid area in the graph editor
- Offer zooming and panning in the grid view via the mouse inputs
- Host and display multiple setting nodes of different types
- Offer a context menu inside the graph editor to add, modify and remove setting nodes instantly
- Offer a mechanic to connect nodes visually and logically via ports and connection noodles
- Copy and clone one or multiple nodes in the graph editor
- Support editing for multiple nodes of same type
- Select and move one or multiple nodes at once

Building a graph editor from scratch is out of scope for this AIS prototype, due to its complexity and required time investment. For this reason, the open-source framework xNode [Extern-2] is used as the foundation for this graph editing tool. This framework is modified greatly to meet the listed graph editor requirements and to be compatible with this AIS prototype in the first place. Due to the vast amount of modifications, as well as it not being relevant for the subject of this thesis, no code snippets are further examined regarding this framework and graph editor.

Figure 5-6 represents the graph editing tool available in this AIS prototype, covering each graph editing feature highlighted in the beginning of this chapter. On the left-hand side, an asset management/creation side panel is displayed, listing each available setting type available to be hosted in the graph editor, with buttons per list item offering functionalities, such as:

- Defining the file path to store newly created settings persistently
- Create and add a new setting to the graph and save the file in the pre-defined persistent file path
- Search for already existing settings in the Unity project and add them to the graph

The topmost tab buttons of the side panel change its contents to:

- Total nodes listed in the graph in addition to the Unity Inspector rendering the selected setting nodes within the graph
- Asset management and creation panel, as already described and shown in Figure 5-6
- Custom code generation panel

The top and rightmost tab buttons outside the side-panel area indicate the node coverage, toggling the display of nodes and node contents, simplifying the graph editor’s view.

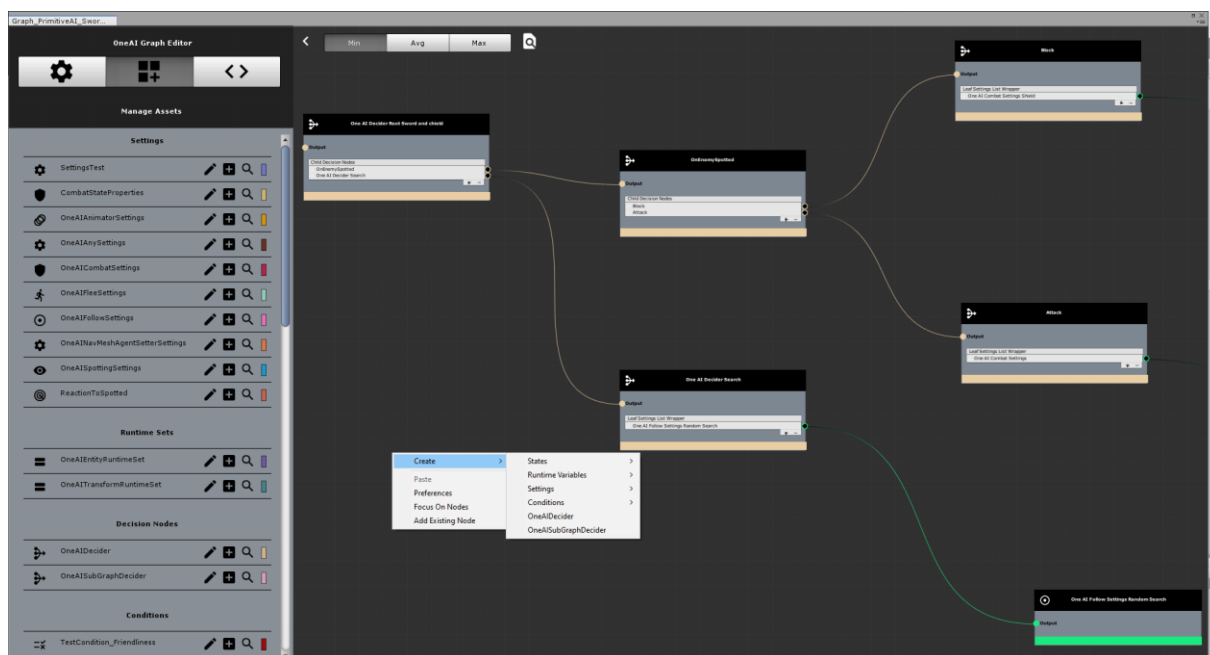


Figure 5-6: OneAI Graph Editor Overview

Due to performance reasons in the Unity Editor, node contents, ports, connections and icons fade away if the graph is zoomed out at a certain distance Figure 5-7. The reason for the performance issue is probably rooted in xNode, combined with ScriptableObjects being rendered individually in Unity’s Editor GUI.



Figure 5-7: OneAI Graph Editor Zoom Fade

Debugging visual tree traversal in the graph editor is partly implemented and can be activated if a Scene is run with a selected Decision System Component in the Unity Editor. This feature is work in progress but offers a basic form of visualization for validated (green) and invalidated (red) settings, illustrated in Figure 5-8.



Figure 5-8: OneAI Graph Editor Visual Tree Traversal

5.8 Code Generation

The code generation tool in this AIS prototype offers custom script creation for:

- Actions
- Action Settings
- Blackboards
- OneAI-Engines
- Descriptors
- Conditions

Figure 5-9 illustrates the code generation panel found in the graph editor and offers the AI programmer to set custom names, as well as folder paths for the automatically generated script files. The generation is triggered by pressing the respective button below the naming fields.

Figure 5-10 showcases the list of template files including the boilerplate code used for the automated code generation, expanding the core AIS logic for new AI behavior. The contents of each template file have special markings like `#SCRIPT_NAME_CONDITION#`, which get replaced by custom set name fields (see Figure 5-9) via file streaming (read and copy template) and string replacement (replace markings with set names).



Figure 5-9: OneAI Code Generator Panel

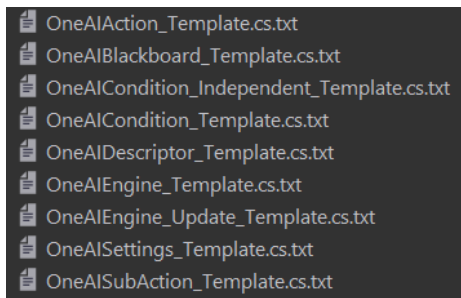


Figure 5-10: OneAI Code Generator Template Files

Although the custom named script creation for the listed types works, this feature is work in progress. The current implementation mainly must be tested for bugs and needs additional and more sophisticated script generation options.

Since this code generator is usable as is for any AI programmer, the issue regarding bad maintainability caused by rewriting boilerplate code is not (entirely) present.

6 Prototype Evaluation

To validate if this prototype meets the requirements defined in Chapter 2 in practice, the evaluation process explained in Section 4.1 is applied on it.

The initial git repository of this prototype contains the wrapper Unity project with the core AIS modules and testing environments together, making it difficult to extract the AIS to other projects with different (newer) Unity versions. For that reason, the core AIS modules are extracted into a separate git repository, which at this stage can be used as a git submodule for new or existing Unity projects.

Evaluation Process: Step 1

Integrating the AIS prototype as a git submodule for the previously set-up Unity evaluation project mentioned in Section 4.1 causes no issues and is compatible with the newer Unity version 2020.3.19f1, with no limitations in regard to OSs. Though the used external package NaughtyAttributes does not appear to work in this testing project. Upon further investigation, it seems like the other AI asset packs collide with this framework, since no issues appear in a new empty Unity project with the same version as the one used for the AI evaluation.

Evaluation Process: Step 2 & 3

Step 2 and 3 of the evaluation processes for this prototype are already established in Chapter 5, describing the modified and mixed AI patterns used in this system.

The result being, that in theory this AIS provides scalability in the form of expandable and inheritable core modules, with improvement and refactoring needed for the Decider settings and the Decision System. Optimal performance is ensured by automatically creating hash values for type mapping and splitting the decision-making algorithms from the action execution implementations. The maintainability is provided by a graph editor in combination with ScriptableObject settings to be (re)used in multiple AI behavior graphs.

The Unity wrapper project provided in the attached USB device contains this AIS prototype and offers example AI use cases (and behaviors), such as fighting, fleeing, following and spotting.

Evaluation Process: Step 4

Listing 6-1 represents the custom action needed for the evaluation process. Line 70 calls the in Listing 4-1 defined performance waster method, used in all AI asset pack evaluation tests. Line 72-77 reads from and writes to the instance-based runtime *CustomDataContainer* displayed in Listing 4-2, by incrementing the local (per instance) and global action calls, as it is defined in the evaluation process.

```

68 ^
69 |
70 |     EvalHelper.WastePerformance();
71 |
72 |     CustomDataContainer dataContainer = blackboard.oneAICustomDescriptor.customDataContainer;
73 |
74 |     dataContainer.actionsCounter++;
75 |     dataContainer.actionsName = this.CustomEngine.name + " Action " + dataContainer.actionsCounter;
76 |
77 |     dataContainer.customSharedDataComponent.totalGlobalActionsCount++;
78 | }
    
```

Listing 6-1: OneAI Benchmark Test Action

Listing 6-2 showcases the custom Condition code, calling the waste performance method in line 16 and similar to the testing action class reads from and writes to the *CustomDataContainer* via a Descriptor Component. Finally incrementing the runtime local (per instance) and global decision calls, starting from line 18 and ending in line 25. Line 27 returns with a chance of 50% either a successful or failed result object.

```

8      public class OneAICustomCondition : OneAIBaseConditionForDescriptors<OneAICustomDescriptor>
9      {
10         [Range(0, 2)]
11         public float minRandomVal = .5f; // 0.5
12
13         // Frequently called 0+23 usages
14         public override OneAIConditionResult IsConditionMetFor
15             (OneAIEntity targetOneAIEntity, OneAIEntity conditionsRequesterEntity)
16         {
17             EvalHelper.WastePerformance();
18
19             OneAICustomDescriptor customDescriptor = targetOneAIEntity.GetDescriptorsByHash<OneAICustomDescriptor>
20                 (this.DescriptorTypeSupportedHash).FirstOrDefault();
21             CustomDataContainer dataContainer = customDescriptor.customDataContainer;
22
23             dataContainer.decisionsCounter++;
24             dataContainer.decisionsName = customDescriptor.name + " Decision " + dataContainer.decisionsCounter;
25
26             dataContainer.customSharedDataComponent.totalGlobalDecisionsCount++;
27
28             return new OneAIConditionResult(success: Random.Range(0f, 1f) > minRandomVal);
29         }
30     }
    
```

Listing 6-2: OneAI Benchmark Test Condition

Figure 6-1 illustrates 7 Decider settings (beige striped nodes) on the left-hand side, setup to behave like selector composites (see Section 3.3) and 8 action settings (green striped nodes) on the right-hand side, setup to dispatch any previous running action and add forever repeating actions if chosen by the Decision System. A zoomed-out view of the balanced decision tree is fully visible in Figure 6-2, showcasing all 16 action settings required for the test case.

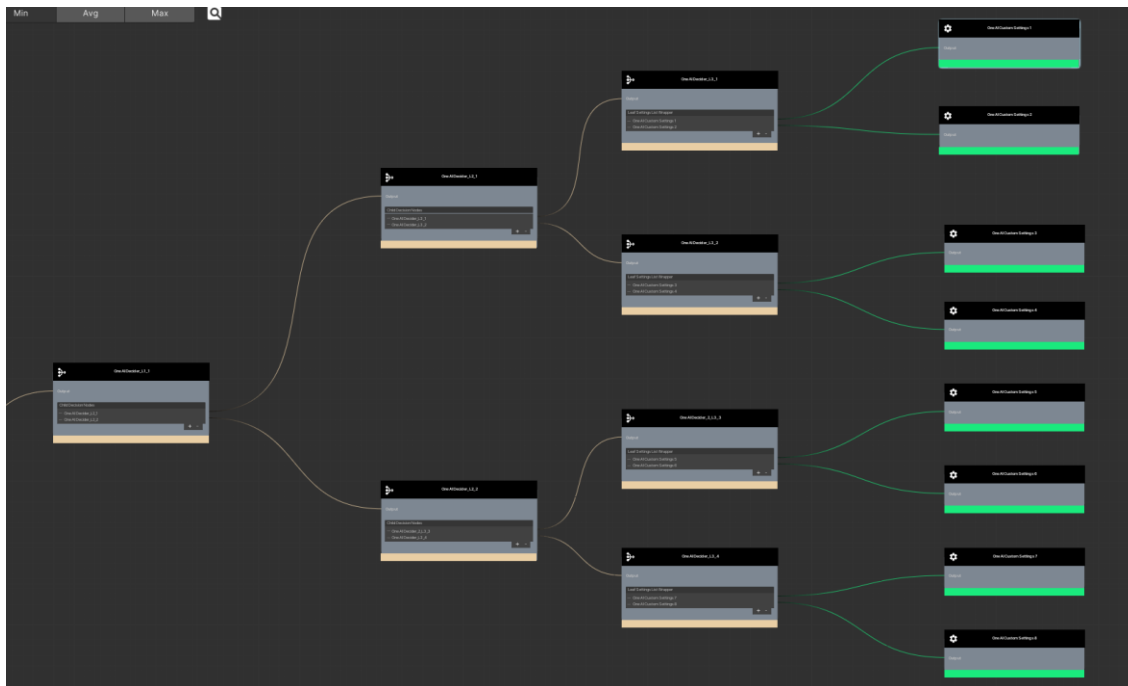


Figure 6-1: OneAI Benchmark Graph Section

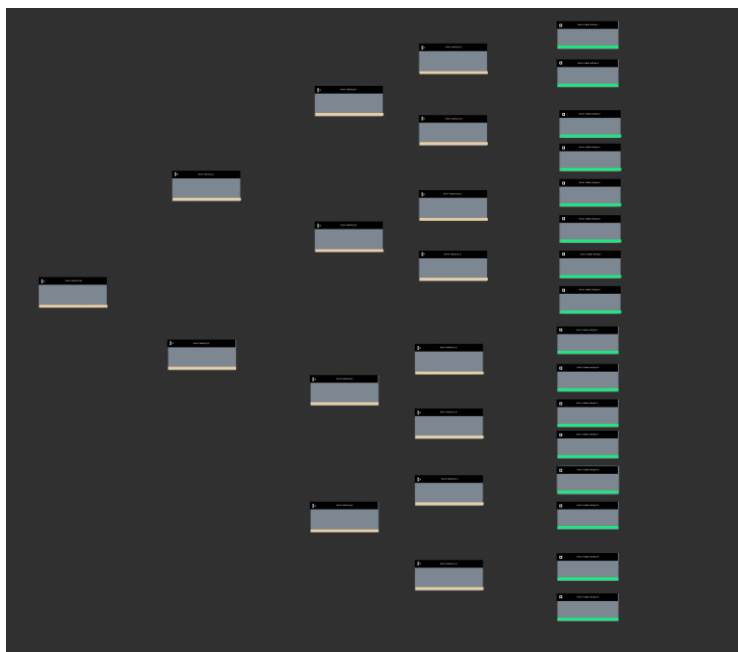


Figure 6-2: OneAI Benchmark Graph Overview

While building the AI behavior in this graph, it is apparent that the graph editor performance is worse, than the node graphs implemented in the evaluated asset packs from Chapter 4. As mentioned in Section 5.7, this is probably caused by either rendering each ScriptableObject individually or the integrated open-source framework xNode. Furthermore, renaming nodes in this graph editor occasionally causes the Unity Editor to crash, which is not reproducible in Unity 2018.4.0f1. Other than that, building the testing AI behavior is fairly simple and fast, due to the supported editing and cloning of multiple selected settings at once. In conclusion, the maintainability of the node editor graph leaves room for improvement but does not fail building a working AI behavior in a swift manner.

ForEach custom action type a respective custom OneAI-Engine is required to be added as a Component to the *OneAIEntity* GameObject. This can lead to maintainability issues if it is not clear to the AI designer, which and how many OneAI-Engines are required by the given AI behavior graph. For that reason, a Unity Editor notification system, indicating missing or required OneAI-Engine types is necessary for the next development iteration of this AIS prototype.

This AIS prototype separates the action execution system from the decision-making system, unlike the evaluated asset packs in Chapter 4 and the investigated AI patterns in Chapter 3. This raises the question in how often should the decision-making algorithm be called?

There is no general representation for every game genre, indicating how often the decision-making process must or will be called at a given frequency. Calling the decision-making algorithm only once or every frame is unlikely to happen, since AI agents oftentimes remain in a certain state for a certain amount of time before changing to another one, repeating the process.

For that reason, the custom Decision System illustrated in Listing 6-3 is created, calling the decision-making algorithm every 100ms with either a chance of 50% or if no action setting has been chosen (see line 18 - 25). As a result, 2 distinctive benchmark test cases are created, one splitting the Decision System from the action execution and one traversing the tree each frame to determine the performance cost differences.

```

8      public class OneAICustomDecisionSystem : DecisionSystem
9      {
10         public float secondsToWaitForRandomCallAttempt = .1f; // Unchanged
11
12         protected override void OnEnable()
13         {
14             base.OnEnable();
15             this.StartCoroutine(routine:this.CallDecisionAtRandom());
16         }
17
18         // Frequently called 1 usage new *
19         private IEnumerator CallDecisionAtRandom()
20         {
21             while (this.isActiveAndEnabled)
22             {
23                 this.makeDecision |= Random.Range(0f, 1f) > .5f || this.oneAIEntityToDecide.currentStates.Count < 1;
24                 yield return new WaitForSeconds(this.secondsToWaitForRandomCallAttempt);
25             }
26         }
27
28         // Event function Nigg
29         private void Update()
30         {
31             this.Tick(this.nodeGraph.rootDecider);
32         }
33     }

```

Listing 6-3: OneAI Benchmark Test Decision System

Splitting the decision-making algorithm from the action execution system results as expected in more actions than decisions being called (see Figure 6-3). The total number of executions does not grow linear for each newly added agent, but rather decreases over time. Furthermore, it appears that the action and decision calls would cross if the graph were to be extrapolated, possibly caused by more random decisions being called based on the implemented arbitrary decision algorithm (see Listing 6-3).

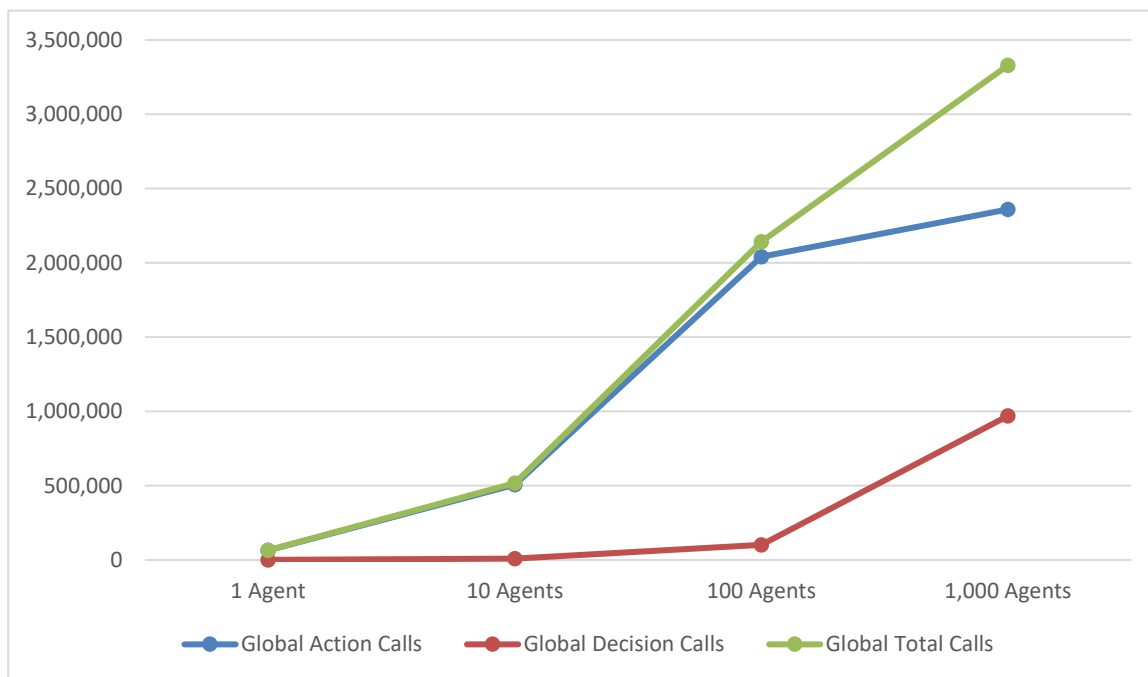


Figure 6-3: OneAI Benchmark Method Calls for 30s (Split Systems)

Figure 6-4 visualizes a linear increasing ms rate per added agent, with the maximum being 4,5ms per frame in the extreme case of 1,000 agents in one Scene at once. This proves, that very little CPU time is wasted when separating decision-making from action execution, even if the decision-making algorithms are potentially called every 100ms, as it is the case in this implementation (see Listing 6-3).

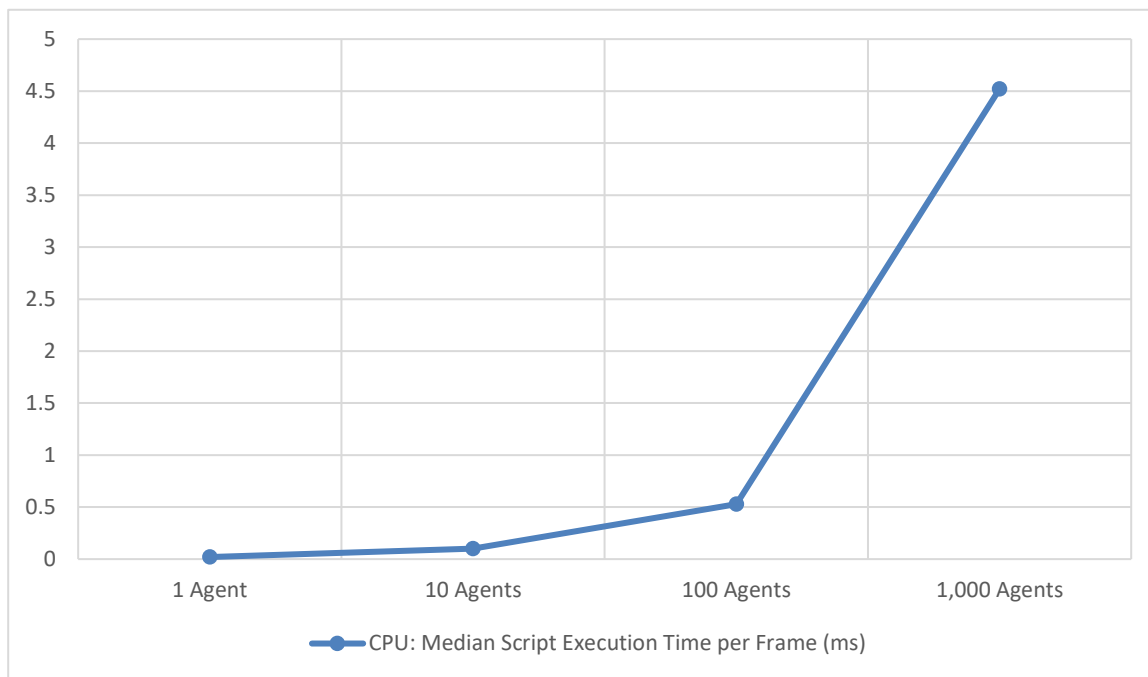


Figure 6-4: OneAI Benchmark CPU Usage (Split Systems)

The values of the **GCAF** column visible in Table 6-1 showcase a linear expansion of memory usage per added agent (each row is multiplied by the factor 10). Due to the fact that the decisions are called at random and thus unpredictable times, the category **GCUM** displays an inconsistent growth relative to the increasing number of agents. As seen in the last column **OC**, each newly added agent represents 6 native objects.

	Memory: GCAF	Memory: GCUM	Memory: OC
1 Agent	134B	17.7MB	6,210
10 Agents	1.4KB	15.3MB – 16.7MB	6,210
100 Agents	12.8KB	16.7MB – 20.1MB	6,750
1,000 Agents	124.5KB	28.9MB – 39.1MB	12,150

Table 6-1: OneAI Benchmark Memory Usage per Frame (Split Systems)

As expected and observed in the previous evaluated AI asset packs (see Section 4.2, 4.4 and 4.5) the decision-making calls overrule the action calls if both systems are not separated from each other (see Figure 6-5). The total number of action calls stagnates and does not reach as high as demonstrated in Figure 6-3, probably due to more performance being wasted on redundant decision calls in addition to less captured frames (140 missing) caused by the restricted testing time of 30s, illustrated in Figure 6-6.

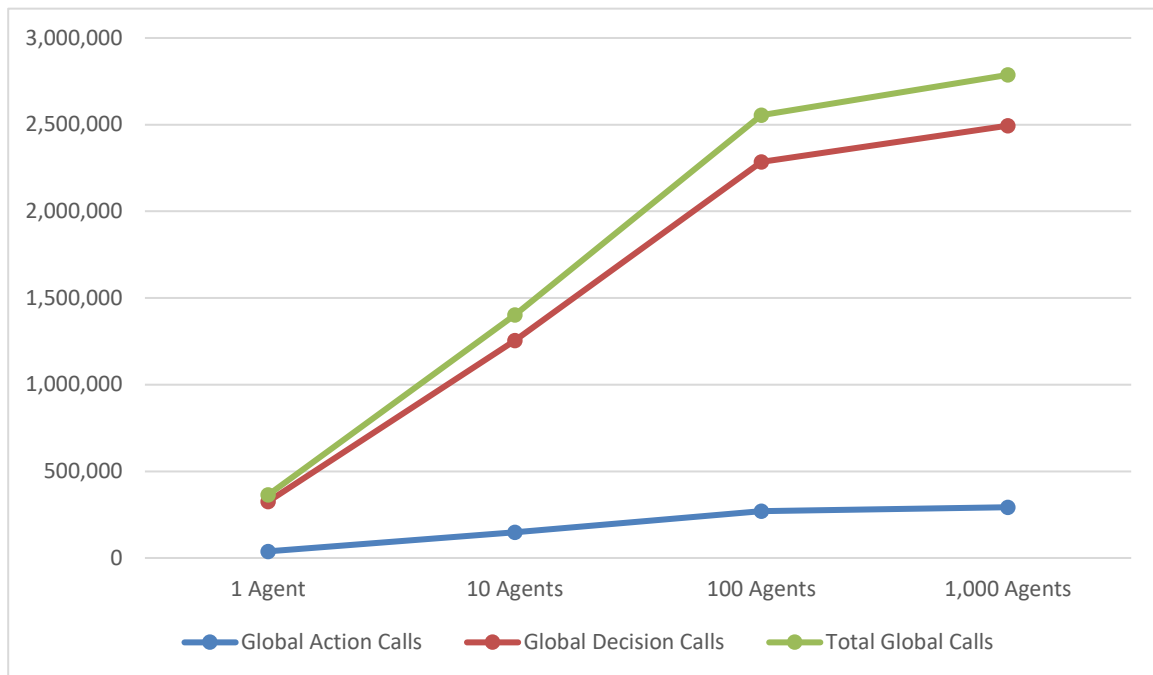


Figure 6-5: OneAI Benchmark Method Calls for 30s (Joined Systems)



Figure 6-6: OneAI Benchmark CPU Profiler (1,000 Agents & Joined Systems)

Figure 6-7 displays a linear growth of the median ms rate per frame foreach added agent to the benchmark test Scene, with the difference to Figure 6-4 being that overall more ms are needed for the decision-making algorithm than the action execution. The extreme high cost of roughly 50ms for 1,000 agents is probably caused by the inefficient decision-making switch statement and the repeating termination and initialization of actions for each frame, concluding that it is not recommended to make a decision call every frame with this AIS prototype.

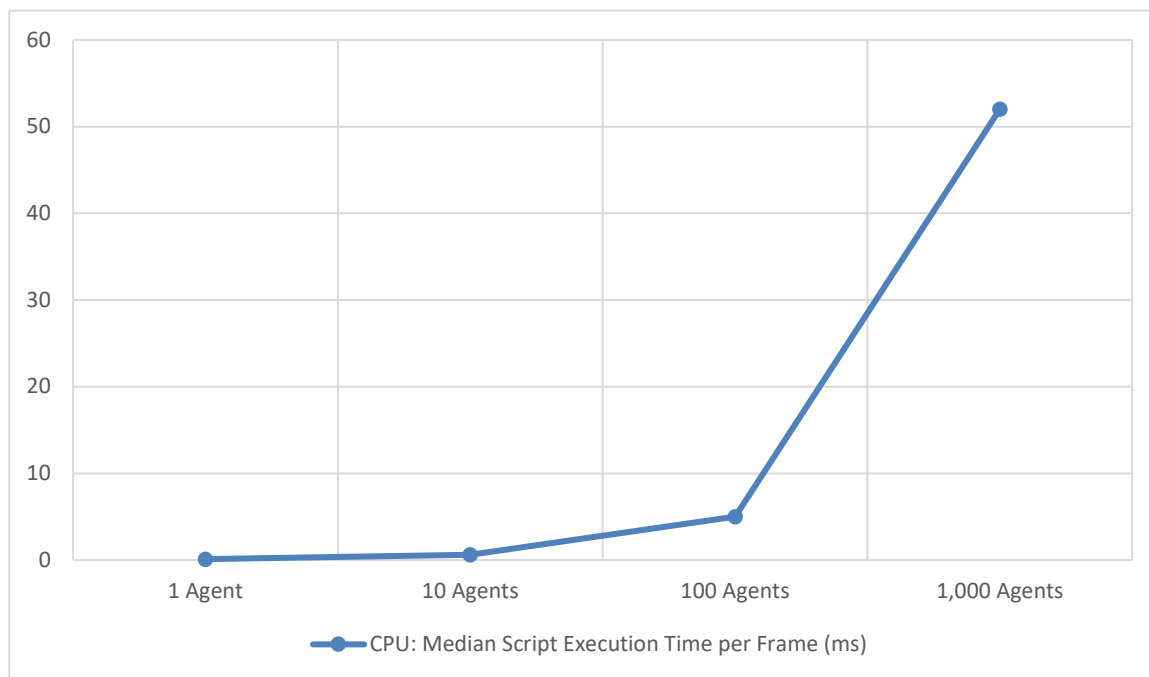


Figure 6-7: OneAI Benchmark CPU Usage (Joined Systems)

Column **GCAF** of Table 6-2 showcases for the first 1 to 10 agents a large jump in memory usage but appears to stagnate or at least is close to being linear in growth for the agent intervals of 10 to 1,000. Similarly, the expansion of memory usage per added agent displayed in the category **GCUM** appears to decrease, especially visible between 100 and 1,000 agents. As for the column **OC**, each agent round about represents 6 native objects, equal to the observations made in Table 6-1.

	Memory: GCAF	Memory: GCUM	Memory: OC
1 Agent	2.0KB – 10KB	16.8MB – 17.5MB	6,200
10 Agents	40.6KB – 71.6KB	16.9MB – 18.4MB	6,240
100 Agents	482.7KB – 600KB	19.2MB – 22.7MB	6,780
1,000 Agents	5.4MB – 5.6MB	31.1MB – 43.2MB	12,180

Table 6-2: OneAI Benchmark Memory Usage per Frame (Joined Systems)

In conclusion, the benchmark tests prove that splitting the decision-making algorithms from the action executions is drastically faster and efficient compared to when convoluting both systems. This prototype appears to manage memory in a clean manner and additionally provides a stable and high FPS rate if the decision-making algorithms are treated separately but not if executed each frame as it is done in common asset packs and AI patterns (see Chapter 3 and 4).

7 Conclusion and Outlook

This thesis highlights issues of the established AI frameworks (see Chapter 3) and asset packs (see Chapter 4) regarding the requirements for a generic AIS defined in Chapter 2 and offers solutions to overcome those problems in Section 3.10 and 4.7, finally implementing them in the AIS prototype in Chapter 5.

As proven by the AIS prototype benchmark results in Chapter 6, the combination of multiple (modified) AI frameworks is the key to build a scalable, performant and maintainable generic AIS.

Overall, this **prototype meets the requirements of a generic AIS, unless the decisions must be executed each frame**. This prototype is capable of being integrated in (new) game projects of any game genre using the Unity3D Engine, although there are still some *unfinished features* and *code parts* that *must be refactored* in the next development iterations, being:

- Generate template scripts via a GUI
- Debug AI decisions and running actions in the graph visually
- Improve the graph editor's performance and stability, by either replacing or further improving the open-source framework xNode
- Refactor the Decision System and Decider settings to fix the scalability issue regarding custom decision-making algorithms when inheriting said classes, as mentioned in Section 5.1.4 and 5.5
- Build a notification system, indicating which OneAI-Engines must be added to an *OneAIEntity* when using the respective graph for maintainability reasons
 - Additionally offer an auto fill option, which adds the OneAI-Engine Components automatically to the *OneAIEntity* GameObject
- Offer special Condition or Decider settings, which implement the Fuzzy Logic pattern of Section 3.6
- Build a test AI behavior, that implements the GOB pattern as described in Section 3.5

Acknowledgments

Special thanks to B. Sc. **Fabio Anthony** for giving me the opportunity to combine my passion and my project with this bachelor thesis, always supporting me with great ideas and references.

Most meetings we had, went over the estimated time because we went down the rabbit hole talking excited about AI, the gaming world and other interesting topics.

Thanks Fabio, for your support!

Appendix A - List of Figures

Figure 3-1: Simple FSM Example.....	9
Figure 3-2: Simple Binary Decision Tree Example	10
Figure 3-3: Simple Non-Limited Decision Tree Example	11
Figure 3-4: Simple Behavior Tree Example.....	12
Figure 4-1: BD Graph UI Excerpt.....	23
Figure 4-2: BD Benchmark Testing Graph Fragment.....	24
Figure 4-3: BD Benchmark Testing Graph Entire View.....	25
Figure 4-4: BD Benchmark Method Calls for 30s	25
Figure 4-5: BD Benchmark CPU Usage	26
Figure 4-6: BD Memory Profiler Excerpt (1,000 Agents)	27
Figure 4-7: EA3 AI Behavior Setup.....	28
Figure 4-8: RVSAI Node Explanation [29]	30
Figure 4-9: RVSAI Test AI Setup	30
Figure 4-10: RVSAI Test Graph Prefab.....	31
Figure 4-11: RVSAI Global Variables	31
Figure 4-12: RVSAI Load Balanced AI Setup.....	33
Figure 4-13: RVSAI Non-Load Balanced AI Setup	33
Figure 4-14: RVSAI Benchmark Test AI Behavior.....	34
Figure 4-15: RVSAI Benchmark Method Calls for 30s (Non-Load Balanced).....	35
Figure 4-16: RVSAI Benchmark CPU Usage (Non-Load Balanced).....	35
Figure 4-17: RVSAI Benchmark Method Calls for 30s (Load Balanced).....	37
Figure 4-18: RVSAI Benchmark CPU Usage (Load Balanced)	38
Figure 4-19: FSMAIT Graph Test.....	40
Figure 4-20: FSMAIT Editor Bug.....	40

Figure 4-21: FSMAIT Benchmark Setup - Any State.....	42
Figure 4-22: FSMAIT Benchmark Setup - Classic	42
Figure 4-23: FSMAIT Benchmark Method Calls for 30s (Classic Setup).....	43
Figure 4-24: FSMAIT Benchmark CPU Usage (Classic Setup).....	44
Figure 4-25: FSMAIT CPU Profiler Excerpt (1,000 Agents & Classic Setup).....	44
Figure 4-26: FSMAIT Benchmark Method Calls for 30s (Any State Setup)	46
Figure 4-27: FSMAIT Benchmark CPU Usage (Any State Setup).....	47
Figure 5-1: OneAI Base Action Settings.....	57
Figure 5-2: OneAI Branch-Leaf Decider	58
Figure 5-3: OneAI Branch Decider	58
Figure 5-4: Generic Mapping Issue.....	60
Figure 5-5: OneAI Decision System Component.....	75
Figure 5-6: OneAI Graph Editor Overview	77
Figure 5-7: OneAI Graph Editor Zoom Fade	78
Figure 5-8: OneAI Graph Editor Visual Tree Traversal	78
Figure 5-9: OneAI Code Generator Panel	79
Figure 5-10: OneAI Code Generator Template Files	80
Figure 6-1: OneAI Benchmark Graph Section.....	83
Figure 6-2: OneAI Benchmark Graph Overview	83
Figure 6-3: OneAI Benchmark Method Calls for 30s (Split Systems)	86
Figure 6-4: OneAI Benchmark CPU Usage (Split Systems)	87
Figure 6-5: OneAI Benchmark Method Calls for 30s (Joined Systems)	88
Figure 6-6: OneAI Benchmark CPU Profiler (1,000 Agents & Joined Systems).....	88
Figure 6-7: OneAI Benchmark CPU Usage (Joined Systems).....	89

Appendix B - List of Listings

Listing 4-1: Performance Heavy Method Simulation	21
Listing 4-2: Custom Data Container	21
Listing 4-3: EA3 RBS Code Snippet.....	29
Listing 4-4: RVSAI Variable Bool Provider.....	32
Listing 4-5: RVSAI Type Conversion.....	32
Listing 4-6: FSMAIT Benchmark Custom Action.....	41
Listing 4-7: UAIS RBS Code Snippet.....	49
Listing 5-1: OneAI Base Condition Method	54
Listing 5-2: OneAI State Condition	54
Listing 5-3: OneAI Base Condition For Descriptors.....	55
Listing 5-4: OneAI Health Descriptor Condition.....	55
Listing 5-5: OneAI Tree Traversal Method	59
Listing 5-6: OneAI Decisions Enum	59
Listing 5-7: OneAI Base Engine	61
Listing 5-8: OneAI Base Settings For Engine.....	61
Listing 5-9: OneAI Automatic Type To Hash Generation.....	62
Listing 5-10: OneAI Action Wrapper Interface	64
Listing 5-11: OneAI Base Action.....	64
Listing 5-12: OneAI Base Action Abstract Methods	65
Listing 5-13: OneAI Base Engine Run Actions	65
Listing 5-14: OneAI Base Engine Wrapper Interface	66
Listing 5-15: OneAI Base Engine Header.....	66
Listing 5-16: OneAI Base Engine Action Queues	67
Listing 5-17: OneAI Base Engine Tick Method	67

Listing 5-18: OneAI Follow Blackboard Example	68
Listing 5-19: OneAI Follow Engine Example	68
Listing 5-20: OneAI Follow Engine Tick Usage	68
Listing 5-21: OneAI Tick In Update Method.....	68
Listing 5-22: OneAI Follow Engine Action Creation	69
Listing 5-23: OneAI Base Engine EDA	70
Listing 5-24: OneAI Health Descriptor Example.....	70
Listing 5-25: OneAI Decision System Tick Method.....	71
Listing 5-26: OneAI Decision System Filter Deciders	72
Listing 5-27: OneAI Decision System Transfer Setting	73
Listing 5-28: OneAI Decision System Update Example.....	73
Listing 5-29: OneAI Entity Fields.....	74
Listing 5-30: OneAI Entity Hash To Engine Method	74
Listing 5-31: OneAI Entity Get Engine By Hash.....	74
Listing 5-32: OneAI Entity Hash To Descriptor Method.....	75
Listing 6-1: OneAI Benchmark Test Action	81
Listing 6-2: OneAI Benchmark Test Condition	82
Listing 6-3: OneAI Benchmark Test Decision System	85

Appendix C - List of Tables

Table 1-1: Unity3D Keywords	6
Table 4-1: Profile Moduler Explanation	22
Table 4-2: BD Benchmark Memory Usage per Frame	27
Table 4-3: RVSAI Benchmark Memory Usage per Frame (Non-Load Balanced)	36
Table 4-4: RVSAI Benchmark Memory Usage per Frame (Load Balanced)	38
Table 4-5: FSMAIT Benchmark Memory Usage (Classic Setup)	45
Table 4-6: FSMAIT Benchmark Memory Usage per Frame (Any State Setup).....	47
Table 6-1: OneAI Benchmark Memory Usage per Frame (Split Systems)	87
Table 6-2: OneAI Benchmark Memory Usage per Frame (Joined Systems).....	89

Appendix D - List of External Frameworks





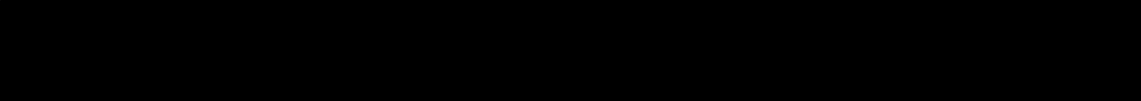
- [Extern-1] D. Rizov & N. Grzonkowski, “GitHub: NaughtyAttributes,” [Online]. Available: <https://github.com/niggo1243/NaughtyAttributes>. [Accessed 10 2021].
- [Extern-2] T. Brigsted, “GitHub: xNode,” [Online]. Available: <https://github.com/Siccity/xNode>. [Accessed 10 2021].

Appendix E - List of Required Tools

- [Tool-1] Microsoft, “Microsoft: C# documentation,” [Online]. Available: <https://docs.microsoft.com/en-us/dotnet/csharp/>. [Accessed 10 2021].
- [Tool-2] Unity3D, “Unity,” [Online]. Available: <https://unity.com/>. [Accessed 10 2021].
- [Tool-3] L. Torvalds, “Git,” [Online]. Available: <https://git-scm.com/>. [Accessed 10 2021].
- [Tool-4] GitLab, “GitLab,” [Online]. Available: <https://gitlab.com/gitlab-org/gitlab>. [Accessed 10 2021].

Bibliography

- [1] M. Buckland, Programming Game AI by Example, 1. ed., USA: Wordware Publishing Inc, 2005.
- [2] Unity3D, “Unity Manual: GameObjects,” 1 8 2017. [Online]. Available: <https://docs.unity3d.com/Manual/GameObject.html>. [Accessed 11 2021].
- [3] Unity3D, “Unity Manual: Prefabs,” 31 7 2018. [Online]. Available: <https://docs.unity3d.com/Manual/Prefabs.html>. [Accessed 11 2021].
- [4] Unity3D, “Unity Manual: ScriptableObject,” 15 10 2018. [Online]. Available: <https://docs.unity3d.com/Manual/class-ScriptableObject.html>. [Accessed 11 2021].
- [5] Unity3D, “Unity Manual: Introduction to components,” [Online]. Available: <https://docs.unity3d.com/Manual/Components.html>. [Accessed 11 2021].
- [6] Unity3D, “Unity Manual: Important Classes - MonoBehaviour,” [Online]. Available: <https://docs.unity3d.com/Manual/class-MonoBehaviour.html>. [Accessed 11 2021].
- [7] Unity3D, “Unity Manual: Scenes,” [Online]. Available: <https://docs.unity3d.com/Manual/CreatingScenes.html>. [Accessed 11 2021].
- [8] Unity3D, “Unity Manual: The inspector window,” 1 2020. [Online]. Available: <https://docs.unity3d.com/Manual/UsingTheInspector.html>. [Accessed 11 2021].
- [9] Unity3D, “Unity Asset Store: AI Search,” [Online]. Available: <https://assetstore.unity.com/?q=AI&orderBy=1>. [Accessed 10 2021].
- [10] A. J. Champanard, AI Game Development: synthetic creatures with learning and reactive behaviors, 2. ed., USA: New Riders Publishing, 2004.
- [11] I. Millington, AI for Games, 3. ed., Boca Raton: CRC Press: Taylor & Francis Group, 2019.
- [12] S. Huang, “FreeCodeCamp What is Big O Notation Explained,” 16 1 2020. [Online]. Available: <https://www.freecodecamp.org/news/big-o-notation-why-it-matters-and-why-it-doesnt-1674cfa8a23c/>. [Accessed 10 2021].

- [13] R. C. Martin, Clean Code: a handbook of agile software craftsmanship, 6. ed., Stoughton, Massachusetts: Pearson Education Inc, 2009.
- [14] Unity3D, “Unity Asset Store: Behavior Search,” [Online]. Available: <https://assetstore.unity.com/?q=behavior&orderBy=1>. [Accessed 10 2021].
- [15] 
- [16] 
- [17] 
- [18] 
- [19] 
- [20] N-Studios, “Unity Asset Store: Ultimate AI System,” [Online]. Available: <https://assetstore.unity.com/packages/tools/ai/ultimate-ai-system-187081>. [Accessed 10 2021].
- [21] Catsoft Works, “Unity Asset Store: Behavior (Game Creator 1),” [Online]. Available: <https://assetstore.unity.com/packages/tools/ai/behavior-game-creator-1-141443>. [Accessed 10 2021].
- [22] DDS Games, “Unity Asset Store: Dynamic AI,” [Online]. Available: <https://assetstore.unity.com/packages/tools/animation/dynamic-ai-118275>. [Accessed 10 2021].
- [23] Walker Boys Studio, “Unity Asset Store: AI Behavior,” [Online]. Available: <https://assetstore.unity.com/packages/tools/ai/ai-behavior-3442>. [Accessed 10 2021].

[24] M. Desjardins, “Unity Asset Store: Breadcrumb Ai,” [Online]. Available: <https://assetstore.unity.com/packages/tools/ai/breadcrumb-ai-18364>. [Accessed 10 2021].

[25] Unity3D, “Unity Manual: The profiler window,” 1 2020. [Online]. Available: <https://docs.unity3d.com/Manual/ProfilerWindow.html>. [Accessed 11 2021].

[26] R. Venables, “Stackoverflow: Cost of common operations for C#?,” 16 5 2009. [Online]. Available: <https://stackoverflow.com/questions/872442/cost-of-common-operations-for-c>. [Accessed 11 2021].

[27]

[28]

[29]

[30]

[31]

[32] M. Warren, “Performance is a Feature: Why is reflection slow?,” 14 12 2016. [Online]. Available: <https://mattwarren.org/2016/12/14/Why-is-Reflection-slow/>. [Accessed 11 2021].